

Package: metacheck (via r-universe)

October 21, 2026

Title Check Research Outputs for Best Practices

Version 0.4.1.2

Date 2026-10-20

Description A modular, extendable system for checking research outputs for best practices using text search, R code, and/or (optional) LLM queries.

License AGPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

Suggests ggplot2, careless, readODS, openxlsx, haven, callr, readxl, data.table, officer, pdftools, SnowballC, yaml, devtools, httptest2, processx, remotes, rmarkdown, testthat (>= 3.0.0), rstudioapi, withr, shiny, shinyjs, shinydashboard, DBI, RSQLite, zip, digest, jsonvalidate, BiocManager

Config/testthat/edition 3

Config/testthat/parallel false

Imports dplyr, tidyr, readr, knitr, quarto, DT, bibtex, ellmer (>= 0.4.2), jsonlite, xml2, curl, httr2 (>= 1.1.0), rvest, rappdirs, tidytext, statcheck, base64enc, gitcreds, roxygen2, jpeg, progress, graphics, grDevices, utils

URL <https://github.com/scienceverse/metacheck>,
<https://scienceverse.github.io/metacheck/>

BugReports <https://github.com/scienceverse/metacheck/issues>

Depends R (>= 4.3.0)

LazyData true

LazyDataCompression xz

Config/Needs/website rmarkdown

Config/roxygen2/version 8.0.0

Config/pak/sysreqs

cmake git make libicu-dev libjpeg-dev libuv1-dev libxml2-dev libssl-dev libx11-dev

Repository <https://scienceverse.r-universe.dev>

Date/Publication 2026-10-21 13:50:20 UTC

RemoteUrl <https://github.com/scienceverse/metacheck>

RemoteRef HEAD

RemoteSha 7b6cd21c3dccbac2a9a2e693aa68150617537bfd

Contents

accuracy	6
add_bib_match	7
aspredicted_info	8
aspredicted_links	8
causal_relations	9
code_abs_path	10
code_extract_py	11
code_extract_qmd_py	12
code_extract_r	13
code_file_refs	13
code_install_packages	14
code_lang	15
code_library_lines	16
code_library_names	16
code_line_stats	18
code_packages	18
code_parse_r	19
code_read	20
code_remove_comments	20
code_setwd	21
collapse_section	22
convert	23
convert_bibr	24
convert_grobid	25
crossref_doi	27
crossref_query	28
datacite_doi	29
dataone_info	30
dataone_links	30
dataverse_file_download	31
dataverse_info	33
dataverse_links	33
dataverse_pat	34
demofile	35
demopaper	35
doi_clean	36
doi_lookup	36
doi_resolves	37

doi_valid_format	38
dryad_auth	38
dryad_file_download	39
dryad_info	41
dryad_links	42
dryad_pat	42
dspace_links	43
dspace7_file_download	44
dspace7_links	44
email	45
emojis	46
export_jasp_html	46
export_mplus_html	47
export_omv_html	47
export_spv_html	48
export_stata_smcl_html	48
extract_eq	49
extract_p_values	50
extract_tests	50
extract_urls	51
fig_image_view	52
figshare_file_download	52
figshare_info	54
figshare_links	55
figshare_pat	55
file_category	56
filetype	57
FLoRA	57
FLoRA_date	58
FLoRA_update	58
format_bib_authors	59
format_ref	59
fsd_info	60
fsd_links	61
get_prev_outputs	61
github_files	62
github_info	63
github_languages	63
github_links	64
github_readme	64
github_repo	65
gitlab_links	66
gitlab_pat	66
gitlab_repo	67
grobid_to_bibr	68
import_jasp	68
import_mplus_output	69
import_omv	70

import_spv	70
import_stata_smcl	71
json_expand	72
lastlog	73
ldbase_info	73
ldbase_links	74
link	75
llm	75
llm_cache	77
llm_cache_clear	77
llm_max_calls	78
llm_max_tokens	78
llm_model	79
llm_model_list	79
llm_reasoning	80
llm_timeout	81
llm_use	82
local_files	82
logger	83
manifest_merge	83
match_reported_output	84
mendeley_info	85
mendeley_links	86
metacheck_cache_info	87
module_help	88
module_info	88
module_install	89
module_installed	90
module_list	90
module_remove	91
module_report	91
module_run	92
module_template	93
module_update	93
openalex_doi	94
openalex_query	96
osf_api_check	97
osf_app	98
osf_cache_clear	98
osf_check_id	99
osf_delay	99
osf_file_download	100
osf_get_all_pages	103
osf_info	104
osf_links	105
osf_pat	105
osf_preprint_list	106
osf_type	107

osf_user_projects	108
paper_id	109
paper_table	109
paper_validate	110
paper_write	111
papers_available	111
papers_load	112
papers_metadata	113
papers_remove	114
path_sanitize	114
plural	115
psycharchives_file_download	116
psycharchives_info	116
psycharchives_links	117
pubpeer_comments	118
rbox_file_download	118
rbox_info	119
rbox_links	119
read	120
read_r_output	121
read_stat_tables	122
ref_table	123
regcheck_compare	123
regcheck_setup_local	125
regcheck_start_local	126
regcheck_stop_local	127
repo_cache_clear	127
repo_cache_dir	128
repo_cache_size	129
repo_info_cache	129
repo_info_cache_clear	130
report	130
report_app	132
report_module_run	132
report_qmd	133
report_repository	134
report_table	135
report_type	136
repro_defined_vars	137
repro_dependencies	138
repro_dependencies_py	139
repro_docker_available	140
repro_file_io	140
repro_install_deps	141
repro_install_deps_docker	142
repro_install_deps_py	143
repro_install_deps_py_docker	144
repro_materialize_layout	144

repro_missing_inputs 145

repro_rewrite_paths 146

repro_run_order 148

repro_run_scripts 149

repro_run_scripts_docker 150

repro_run_scripts_py 152

repro_run_scripts_py_docker 153

repro_write_scripts 154

researchdata4tu_file_download 156

researchdata4tu_info 157

researchdata4tu_links 158

researchdata4tu_pat 159

reshare_file_download 160

reshare_info 161

reshare_links 162

retractionwatch 162

rw_date 163

rw_update 164

scroll_table 164

stat_output_json 165

stat_output_validate 166

stat_output_write 167

stat_results_long 167

stato_type_column 168

stats 169

test_paper 170

text_expand 170

text_search 171

validate 173

zenodo_file_download 174

zenodo_info 175

zenodo_links 176

zenodo_pat 176

zenodo_upload 177

zip_peek_cache_clear 180

Index **181**

accuracy	<i>Accuracy</i>
----------	-----------------

Description

Signal detection values for modules that classify papers as having a feature or not

Usage

accuracy(expected, observed)

Arguments

expected a vector of logical values for the expected values
observed a vector of logical values for the observed values

Value

a list of accuracy parameters

add_bib_match	<i>Match table from bib table</i>
---------------	-----------------------------------

Description

Match table from bib table

Usage

```
add_bib_match(paper, min_score = 50)
```

Arguments

paper a paper or paperlist object
min_score minimal score that is taken to be a reliable match

Value

the paper or paperlist with bib_match table added

Examples

```
## Not run:  
paper <- demopaper()  
paper$bib_match <- NULL # remove existing  
paper2 <- add_bib_match(paper)  
paper2$bib_match  
  
## End(Not run)
```

aspredicted_info	<i>Retrieve info from AsPredicted by URL</i>
------------------	--

Description

Retrieve info from AsPredicted by URL

Usage

```
aspredicted_info(ap_url, id_col = 1, wait = 1)
```

Arguments

ap_url	an AsPredicted URL, or a table containing them (e.g., as created by aspredicted_links())
id_col	the index or name of the column that contains AsPredicted URLs, if id is a table
wait	wait time in seconds

Value

a data frame of information

aspredicted_links	<i>Find AsPredicted Links in Papers</i>
-------------------	---

Description

Find AsPredicted Links in Papers

Usage

```
aspredicted_links(paper)
```

Arguments

paper	a paper object or paperlist object
-------	------------------------------------

Value

a table with the AsPredicted url in the first (href) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
aspredicted_links(psychsci)

## End(Not run)
```

causal_relations	<i>Extract causal relations from sentence(s) via a Hugging Face Space</i>
------------------	---

Description

Sends one or more input sentences to a public Gradio app hosted on Hugging Face (the *lakens-causal-sentences* Space), created based on code by Rasoul Norouzi, retrieves the result via Server-Sent Events (SSE), and returns a tidy data frame with one row per detected cause–effect relation.

Usage

```
causal_relations(
  sentence,
  rel_mode = "auto",
  rel_threshold = 0.5,
  cause_decision = "cls+span",
  timeout = 10,
  verbose = FALSE
)
```

Arguments

sentence	A character vector of one or more sentences to analyze for causal relations.
rel_mode	Relation extraction mode. Options are "auto" (default) or "neural_only".
rel_threshold	Numeric threshold (default 0.5) for deciding whether a relation is considered causal.
cause_decision	Strategy for cause/effect detection. Options: "cls_only", "span_only", or "cls+span" (default).
timeout	Maximum time (in seconds) to wait for the Hugging Face Space to return a result via SSE before aborting. Default is 10.
verbose	Logical; if TRUE, prints diagnostic information (URLs, status codes, event progression). Default FALSE.

Details

The function uses Gradio's two-step queue API: (1) a POST request enqueues the job and returns an event_id; (2) a GET request streams text/event-stream frames until event: complete. Many Gradio apps emit a **double-encoded** completion payload of the form ["<final JSON string>"]. This function unwraps that to obtain the final JSON structure (an array of items containing causal and relations) before parsing.

If a sentence has **no relations**, the output includes one row with cause = NA, effect = NA, and the sentence's causal flag (as returned by the model). For sentences with **multiple relations**, the function returns one row per relation.

Value

A base data.frame with columns:

- sentence (character): the original input sentence,
- causal (logical): whether the sentence is causal per the model,
- cause (character): extracted cause span (or NA),
- effect (character): extracted effect span (or NA).

References

Norouzi, R., Kleinberg, B., Vermunt, J. K., & van Lissa, C. J. (2025). Capturing causal claims: A fine-tuned text mining model for extracting causal sentences from social science papers. *Research Synthesis Methods*, 16(1), 139–156. <https://doi.org/10.1017/rsm.2024.13>

Hugging Face Model Card: rasoultiburg/SocioCausaNet <https://huggingface.co/rasoultiburg/SocioCausaNet>

Examples

```
## Not run:
# Single sentence
df1 <- causal_relations("Smoking causes cancer")
print(df1)

# Multiple sentences (batch)
df2 <- causal_relations(c("Insomnia causes depression.", "Rain leads to flooding."))
print(df2)

# Custom parameters and verbose diagnostics
df3 <- causal_relations(
  sentence = "Stress increases blood pressure.",
  rel_mode = "auto",
  rel_threshold = 0.4,
  cause_decision = "cls+span",
  timeout = 10,
  verbose = TRUE
)
print(df3)

## End(Not run)
```

code_abs_path

Return Absolute Paths

Description

Check code for the presence of absolute paths

Usage

```
code_abs_path(code_text)
```

Arguments

code_text the text of the code, excluding comments

Value

a vector of absolute paths

Examples

```
code_text <- c(
  "file <- 'C:/User/lakens/file.R'",
  "tmp <- '/User/lakens/file.html'",
  "convert(file, tmp)"
)
code_abs_path(code_text)
```

code_extract_py	<i>Convert a Jupyter notebook to code only</i>
-----------------	--

Description

A .ipynb is a JSON document, not a text script: its source lives in the source array of each "code" cell. This concatenates those cells in document order so the result can be checked by the same text-based helpers every other language goes through ([code_remove_comments\(\)](#), [code_library_names\(\)](#), [code_file_refs\(\)](#), ...) — the notebook analogue of [code_extract_r\(\)](#) purling an .Rmd/.qmd.

Usage

```
code_extract_py(file_path = NULL, save_path = NULL, text = NULL)
```

Arguments

file_path path to a .ipynb file
 save_path if NULL, returns a text vector, else a path to save to
 text alternative to file_path, pass the notebook JSON directly

Details

Jupyter is kernel-agnostic, so a notebook's cells may be Python **or** R (or another language); this extracts the source whatever the kernel, and [code_lang\(\)](#) is what reports which language it is. The function is named for the file type it reads, not for a language it assumes.

Markdown and raw cells are dropped (they are prose, not code). Cell boundaries are preserved as blank lines so line counts stay meaningful, and IPython magics (%matplotlib inline) and shell escapes (!pip install x) are dropped: they are not statements in any kernel language and would only ever be noise to the checks downstream.

The OUTPUTS a notebook stores alongside its code are not touched here — see [read_stat_tables\(\)](#), which reads those as result tables the same way it reads a .jasp/.omv archive.

Value

a character vector of source lines (empty when the file has no code cells or is not parseable JSON)

code_extract_qmd_py	<i>Extract Python chunks from a Quarto document</i>
---------------------	---

Description

A .qmd whose declared/inferred engine is Python (see `code_lang()`'s `.qmd_lang()`) cannot go through `code_extract_r()`: `knitr::purl()` only recovers ````{r}` chunks, so a Python-engine document would purl to nothing. This is the .qmd analogue of `code_extract_r()` for that case — concatenating the body of every ````{python}` fence in document order, the same "recover a checkable code file" idea `code_extract_py()` applies to a .ipynb's code cells (chunk boundaries are preserved as a blank line so line counts stay meaningful).

Usage

```
code_extract_qmd_py(file_path = NULL, save_path = NULL, text = NULL)
```

Arguments

<code>file_path</code>	a path to a .qmd file
<code>save_path</code>	if NULL, returns a text vector, else a path to save to
<code>text</code>	alternative to <code>file_path</code> , pass text directly

Details

Unlike `code_extract_r()`, this is a plain text/regex scan, not a real Pandoc/Quarto parse: a fence marker appearing inside a string literal or a displayed (non-executable) code block written with four backticks around a literal ````{python}` would be misread as a real chunk. This mirrors the fence-based approach already used elsewhere in the module (e.g. `code_parse_r()`'s own `^---\\s*$` front-matter check) rather than introducing a new parsing dependency for a single edge case.

Value

a character vector of Python source lines (empty when the document has no Python chunks)

code_extract_r	<i>Convert Rmd/qmd files to R code only</i>
----------------	---

Description

Convert Rmd/qmd files to R code only

Usage

```
code_extract_r(
  file_path = NULL,
  save_path = NULL,
  documentation = 0,
  text = NULL
)
```

Arguments

file_path	a vector of file paths to check
save_path	if NULL, returns a text vector, else a path to save to
documentation	0:2 value to pass to knitr::purl
text	alternative to file_path, pass text directly

Value

a character vector

Examples

```
file_path <- demofile("qmd")
code_text <- code_extract_r(file_path)
```

code_file_refs	<i>Get files referenced in code</i>
----------------	-------------------------------------

Description

Get files referenced in code

Usage

```
code_file_refs(
  code_text,
  lang = c("R", "Python", "SPSS", "SAS", "Stata", "Mplus", "MATLAB"),
  include_writes = FALSE
)
```

Arguments

code_text	the code text for a single file
lang	the language (we only currently handle R, Python, SPSS, SAS, Stata, Mplus, MATLAB)
include_writes	also return files the code <i>writes</i> (R only). Off by default: callers that ask "which referenced inputs are missing from the repository?" (e.g. code_check) must not see a written file as a missing input. repro_file_io() turns this on so a file produced by one script can be recognised as the input another script consumes.

Value

a vector of files that are referenced in the code

Examples

```
code_text <- c(
  'source("functions.R")',
  'a <- "bread"',
  'b <- read.csv("file.csv")'
)
code_file_refs(code_text, "R")
```

code_install_packages *Find install.packages() calls in code*

Description

An `install.packages()` call in analysis code installs software on whoever runs the script, without asking – a portability and security concern raised in issue #398: at minimum it is disruptive (unexpected installs, network access, a package version the author didn't test against), and at worst a security risk if it points at a private/ untrusted repository. Best practice is to document required packages (a README, a `renv.lock/DESCRIPTION`) rather than install them from inside the analysis code itself. This scans the (comment-free) code for `install.packages(...)` calls and returns one row per call, with the argument as written. Only a LIVE call is ever seen here: a commented-out `#install.packages(...)` is invisible to this function by construction, the same way `code_setwd()` never sees a commented-out `setwd()` – both are only ever called on `code_remove_comments()`'s output. R-only (the construct is R's; other languages have their own package managers, which this does not scan).

Usage

```
code_install_packages(code_text)
```

Arguments

code_text	the code text for a single file (character vector), ideally comment-free (as produced by <code>code_remove_comments()</code>)
-----------	--

Value

a data frame with columns `install_packages_call` (the `install.packages(...)` text as written) and `line` (its line number). Empty frame when none are found.

Examples

```
code_text <- c(
  "install.packages('ggplot2')",
  "library(ggplot2)"
)
code_install_packages(code_text)
```

code_lang	<i>Detect Code Language</i>
-----------	-----------------------------

Description

Detects code language used in files, only for languages metacheck currently processes (R, SAS, SPSS, Stata, Mplus).

Usage

```
code_lang(file_name)
```

Arguments

`file_name` a vector of file names

Value

a vector of languages

Examples

```
file_name <- "file.R"
code_lang(file_name)

file_name <- c("file.Rmd", "file.SAS", "file.r", "file.qmd", "file.txt")
code_lang(file_name)
```

code_library_lines	<i>Get Code Library Lines</i>
--------------------	-------------------------------

Description

Returns the lines on which library/require calls exist. This is a helper function for the code_check module.

Usage

```
code_library_lines(
  code_text,
  lang = c("R", "Python", "SPSS", "SAS", "Stata", "Mplus", "MATLAB")
)
```

Arguments

code_text	the code text for a single file
lang	the language (we only currently handle R, Python, SPSS, SAS, Stata, Mplus, MATLAB)

Value

a data frame with columns code and line (the line numbers on which library calls exist, after removing blank lines and comments)

Examples

```
code_text <- c(
  "library(dplyr)",
  "",
  "# this line won't count",
  "library(tidyr)",
  "renv::install('metacheck')"
)
code_library_lines(code_text, "R")
```

code_library_names	<i>Get package names loaded in code</i>
--------------------	---

Description

Extracts the names of the packages/libraries a code file loads. Where `code_library_lines()` only reports the lines on which imports occur (to check they are grouped), this returns the actual package identifiers so they can be catalogued and searched, or written to a requirements.txt.

Usage

```
code_library_names(
  code_text,
  lang = c("R", "Python", "SPSS", "SAS", "Stata", "Mplus", "MATLAB")
)
```

Arguments

code_text	the code text for a single file
lang	the language (R, Python, SPSS, SAS, Stata, Mplus, MATLAB)

Details

For R this captures `library()` / `require()` / `requireNamespace()` (bare or quoted argument), `pacman`'s `p_load()` (which may list several packages at once), namespace-qualified calls (`pkg::fun` / `pkg:::fun`), and `install.packages()` / `renv::install()`. For Python it captures `import a`, `import a as b`, `import a, b`, and `from a.b import c` (recording the top-level package `a`). SAS / SPSS / Stata have no package concept, so an empty frame is returned.

The extraction is regex-based (like the rest of the module) so it degrades gracefully on files that will not parse. It can therefore miss dynamically constructed package names, and does not know which *version* was used — only the names appear in the source.

Where a call takes a variable rather than a package name — `install.packages(pkgs)`, `requireNamespace(pkg)`, `library(pkg, character.only = TRUE)`, `lapply(pkgs, library, ...)` — the variable is resolved to the names it holds when the file assigns it a literal vector (`pkgs <- c("a", "b")`, including a `for (pkg in pkgs) loop` variable), and dropped otherwise. The variable's own name is never reported as a package.

Value

a data frame with columns `package`, `source` (how the package was referenced: `library`, `require`, `requireNamespace`, `p_load`, `namespace`, `install`, or `import`), and `line` (the line number). Rows are unique on `package + source + line`. An empty frame (same columns) when none found.

Examples

```
code_text <- c(
  "library(dplyr)",
  "require('tidyr')",
  "pacman::p_load(ggplot2, readr)",
  "x <- stringr::str_trim(' a ')"
)
code_library_names(code_text, "R")
```

code_line_stats	<i>Get Code Composition Stats</i>
-----------------	-----------------------------------

Description

Get Code Composition Stats

Usage

```
code_line_stats(
  code_text,
  lang = c("R", "Python", "SPSS", "SAS", "Stata", "Mplus", "MATLAB")
)
```

Arguments

code_text	the code text for a single file
lang	the language (we only currently handle R, Python, SPSS, SAS, Stata, Mplus, MATLAB)

Value

list with items total_lines, comment_lines, code_lines, percent_comment, and (Python only) has_docstring

Examples

```
code_text <- c(
  "library(dplyr)",
  "",
  "# this line is a comment",
  "a <- 1"
)
code_line_stats(code_text, "R")
```

code_packages	<i>Distinct packages from a code_check table</i>
---------------	--

Description

The code_check module stores the packages each code file loads as a comma-joined string in its table\$packages column. This returns the sorted, de-duplicated union across a set of those rows — the paper-level dependency list used for the module summary, the manifest code section, and the requirements.txt written into a Psych-DS archive by convert_psychds().

Usage

```
code_packages(packages)
```

Arguments

packages a character vector of comma-joined package strings (e.g. `code_check(...)$table$packages`), or a `code_check` table (data frame with a `packages` column).

Value

a sorted character vector of distinct package names (possibly empty)

Examples

```
code_packages(c("dplyr, ggplot2", "", "dplyr, tidyr"))
```

code_parse_r	<i>Parse code to check for errors</i>
--------------	---------------------------------------

Description

Parse code to check for errors

Usage

```
code_parse_r(file_path = "", text = NULL)
```

Arguments

file_path a vector of file paths to check
text alternative to `file_path`, pass text directly

Value

a data frame with columns `file_path` and `line`

Examples

```
file_path <- demofile("qmd")  
code_parse_r(file_path)
```

code_read	<i>Read code from files</i>
-----------	-----------------------------

Description

Read code from files

Usage

```
code_read(file_path)
```

Arguments

file_path a file path or url to read in

Value

a character vector of the file contents

Examples

```
file_path <- demofile("json")
text <- code_read(file_path)
```

code_remove_comments	<i>Remove comments from code text</i>
----------------------	---------------------------------------

Description

Remove comments from code text

Usage

```
code_remove_comments(
  code_text,
  lang = c("R", "Python", "SPSS", "SAS", "Stata", "Mplus", "MATLAB")
)
```

Arguments

code_text the code text for a single file
lang the language (we only currently handle R, Python, SPSS, SAS, Stata, Mplus, MATLAB)

Value

the code_text minus comment lines

Examples

```
code_text <- c(
  "# this is a comment",
  "",
  "x <- 'And this is code'"
)
code_text_nc <- code_remove_comments(code_text, "R")
```

code_setwd	<i>Find setwd() calls in code</i>
------------	-----------------------------------

Description

A `setwd()` call in analysis code is a portability problem: it hardcodes an assumption about the working directory the code runs in (frequently an absolute path on the author's own machine), mutates global state, and makes a script depend on being run from a particular place. Best practice is to keep the working directory as the caller sets it and use relative paths. This scans the (comment-free) code for `setwd(...)` calls and returns one row per call, with the argument as written. R-only (the construct is R's; other languages have their own, e.g. Stata `cd`, which this does not scan).

Usage

```
code_setwd(code_text)
```

Arguments

<code>code_text</code>	the code text for a single file (character vector), ideally comment-free (as produced by <code>code_remove_comments()</code>)
------------------------	--

Value

a data frame with columns `setwd_call` (the `setwd(...)` text as written) and `line` (its line number). Empty frame when none are found.

Examples

```
code_text <- c(
  "setwd('D:/Dropbox/project')",
  "x <- read.csv('data.csv')"
)
code_setwd(code_text)
```

collapse_section *Make Collapsible Section*

Description

A helper function for making module reports.

Usage

```
collapse_section(
  text,
  title = "Learn More",
  callout = c("tip", "note", "warning", "important", "caution"),
  collapse = TRUE
)
```

Arguments

text	The text to put in the collapsible section; vectors will be collapse with line breaks between (e.g., into paragraphs)
title	The title of the collapse header
callout	the type of quarto callout block
collapse	whether to collapse the block at the start

Details

When `report_type()` is "simple" or "simple_brief", this renders as a plain fenced div with the title as a real bold line of text, not a Quarto callout: Quarto's title/collapse fenced-div attributes are Quarto-specific – plain Pandoc (what `report()`'s simple-mode render uses, see its own docs) passes an unrecognised attribute straight through as a literal (invisible) HTML attribute rather than rendering it as content, and expanding/collapsing needs Bootstrap's collapse JavaScript, which a simple report never includes. The content is always shown inline, uncollapsed. In a "checklist" report it is a <details> box instead, which folds without JavaScript.

Value

text

Examples

```
text <- c("Paragraph 1...", "Paragraph 2...")
collapse_section(text) |> cat()
```

convert

*Convert documents***Description**

Uses grobid or bibr to convert a file to paper format.

Usage

```
convert(
  file_path,
  save_path = ".",
  method = c("auto", "bibr", "grobid", "xml"),
  crossref_lookup = FALSE,
  keep_xml = TRUE,
  ...
)
```

Arguments

<code>file_path</code>	Path to the document file, or a directory of documents
<code>save_path</code>	Path to a directory in which to save the JSON file
<code>method</code>	whether to use bibr, grobid, or xml (grobid_to_bibr) to convert a file (see Details)
<code>crossref_lookup</code>	whether to add the bib_match table from crossref
<code>keep_xml</code>	if the method is grobid, whether to keep intermediate XML files
<code>...</code>	further arguments to pass to <code>convert_bibr</code> , <code>convert_grobid</code> , or <code>grobid_to_bibr</code>

Details

Both bibr and grobid can handle PDF files. Only bibr can convert doc or docx files. Already-converted grobid XML files can be converted to bibr format (set `crossref_lookup=TRUE` to add a `bib_match` table). If the `file_path` is a directory, the method will be xml if any XML files are present, and bibr if only doc or docx files are present.

With method "auto" or "bibr", PDF files are converted with bibr on metacheck's API on SURF Research Cloud (see [convert_bibr](#)) when it is available, unless `api_url`, `api_key` or backend is given, or there are doc or docx files. Otherwise the method "auto" uses a local grobid (<http://localhost:8070>), then a local bibr (<http://localhost:8000>), then the first available server in the online list at <https://www.scienceverse.org/metacheck/convert.json>.

Value

the path to the JSON file

convert_bibr

*Convert documents using bibr***Description**

Converts document files (PDF, DOC, DOCX) to structured JSON using the bibr extraction service. Supports three backends: metacheck's API on SURF Research Cloud ("surf"), which converts PDF files and needs no API key; the Scienceverse platform ("scivrs") which uses a job queue with load balancing; and a self-hosted bibr instance ("selfhosted") for direct API access.

Usage

```
convert_bibr(
  file_path,
  save_path = ".",
  backend = c("auto", "surf", "scivrs", "selfhosted"),
  api_key = NULL,
  api_url = NULL,
  include_figures = FALSE,
  start_page = 1,
  end_page = Inf,
  poll_interval = 2,
  timeout = 600
)
```

Arguments

file_path	Path to the document file, or a directory of documents
save_path	Path to a directory in which to save the JSON file
backend	Which backend to use: "auto" (default) detects from the available API key and URL (see Details), "surf" uses metacheck's API on SURF Research Cloud, "scivrs" uses the Scienceverse platform, "selfhosted" uses a direct bibr API instance.
api_key	API key. For the scivrs backend, a Bearer token starting with sv_, defaulting to the SCIVRS_API_KEY env var. For the selfhosted backend, the server's bearer token if it requires one, defaulting to the BIBR_API_KEY env var (none if unset).
api_url	Base URL of the API. Defaults to the appropriate URL for the selected backend (for surf, the SURF_API_URL env var or https://metacheckserve.metacheck.src.surf-hosted.nl/api/v1/ for selfhosted, the BIBR_API_URL env var or http://localhost:8000/).
include_figures	Whether to include base64-encoded figure images in the output (default FALSE; not for the surf backend)
start_page	First page of the file to extract (default 1; not for the surf backend)
end_page	Last page of the file to extract (default Inf for all pages; not for the surf backend)
poll_interval	Seconds between status polls, scivrs backend only (default 2)
timeout	Maximum seconds to wait for processing (default 600)

Details

When backend = "auto" (the default), the "scivrs" backend is used if api_key is provided or the SCIVRS_API_KEY environment variable is set, and "selfhosted" if api_url is provided or the BIBR_API_URL environment variable is set. Otherwise, "surf" is used.

The "surf" backend sends the PDF to the /convert endpoint of metacheck's API on SURF Research Cloud, whose address can be set with the SURF_API_URL environment variable. It converts only PDF files (at most 50 MB), always whole and without figures, so include_figures, start_page and end_page do not apply. The server deletes the upload once it is converted, and the result an hour later.

A self-hosted bibr server that requires authentication (bibr serve with AUTH_API_KEY set, as on a SURF Research Cloud workspace) gets its bearer token from api_key or the BIBR_API_KEY environment variable; its address can be set with BIBR_API_URL.

Value

Path(s) to the saved JSON file(s)

Examples

```
## Not run:
# Auto-detect backend from environment variables
pdf <- demofile("pdf")
convert_bibr(pdf)

# Explicitly use metacheck's API on SURF Research Cloud
convert_bibr(pdf, backend = "surf")

# Explicitly use Scienceverse platform
convert_bibr(pdf, backend = "scivrs")

# Use self-hosted bibr instance
convert_bibr(pdf, backend = "selfhosted")

# Extract specific pages
convert_bibr(pdf, start_page = 1, end_page = 10)

# Directory of papers
dir <- system.file("demo", package = "metacheck")
convert_bibr(dir, save_path = "results/")

## End(Not run)
```

Description

This function uses a GDPR-compliant public grobid server maintained by Eindhoven Technical University. You can set up your own local grobid server following instructions from <https://grobid.readthedocs.io/> and set the argument `api_url` to its path (probably <http://localhost:8070>). See <https://github.com/grobidOrg/grobid#demo> for other publicly available servers (we cannot guarantee their privacy).

Usage

```
convert_grobid(
  file_path,
  save_path = ".",
  api_url = "https://grobid.hti.ieis.tue.nl",
  start_page = -1,
  end_page = -1,
  consolidate_citations = 0,
  consolidate_header = 0,
  consolidate_funders = 0
)
```

Arguments

<code>file_path</code>	path to the PDF, a vector of paths, or a directory name that contains PDFs
<code>save_path</code>	directory or file path to save to; set to NULL to return the XML directly
<code>api_url</code>	the URL to the grobid server
<code>start_page</code>	the first page of the PDF to read (defaults to -1 to read all pages)
<code>end_page</code>	the last page of the PDF to read (defaults to -1 to read all pages)
<code>consolidate_citations</code>	whether to fix/enhance citations
<code>consolidate_header</code>	whether to fix/enhance paper info
<code>consolidate_funders</code>	whether to fix/enhance funder info

Details

Consolidation of citations, headers, and funders looks up these items in CrossRef or another database to fix or enhance information (see <https://grobid.readthedocs.io/en/latest/Consolidation/>). This can slow down conversion. Consolidating headers is only useful for published papers, and can be set to 0 for work in prep. We recommend you leave these defaults at 0 and use `crossref_lookup = TRUE` when converting from grobid XML to bibr JSON format with the `convert()` function.

Value

XML object

crossref_doi

*CrossRef Info from DOI***Description**

Valid selects for crossref API are:

Usage

```
crossref_doi(
  doi,
  select = c("DOI", "type", "title", "author", "container-title", "volume", "issue",
    "page", "URL", "abstract", "year", "error")
)
```

Arguments

doi	the DOI of the paper to get info for
select	what fields to select from the crossref API

Details

abstract, URL, resource, member, posted, score, created, degree, update-policy, short-title, license, ISSN, container-title, issued, update-to, issue, prefix, approved, indexed, article-number, clinical-trial-number, accepted, author, group-title, DOI, is-referenced-by-count, updated-by, event, chair, standards-body, original-title, funder, translator, published, archive, published-print, alternative-id, subject, subtitle, published-online, publisher-location, content-domain, reference, title, link, type, publisher, volume, references-count, ISBN, issn-type, assertion, deposited, page, content-created, short-container-title, relation, editor

Value

data frame with DOIs and info

Examples

```
doi <- "10.7717/peerj.4375"
## Not run:
# cr_info <- crossref_doi(doi)

## End(Not run)
```

crossref_query	<i>Look up Reference in CrossRef</i>
----------------	--------------------------------------

Description

Look up Reference in CrossRef

Usage

```
crossref_query(
  ref,
  min_score = 50,
  rows = 1,
  select = c("DOI", "score", "type", "title", "author", "editor", "publisher",
    "container-title", "year", "volume", "issue", "page", "URL")
)
```

Arguments

ref	the full text reference of the paper to get info for, see Details
min_score	minimal score that is taken to be a reliable match (default 50)
rows	the maximum number of rows to return per reference (default 1)
select	what fields to select from the crossref API

Details

The argument ref can take many formats. Crossref queries only look for authors, title, and container-title (e.g., journal or book), but extra information doesn't seem to hurt.

- a text reference or fragment
- a bibentry object (authors, title and container will be extracted)
- a vector of text or bibentry objects
- a paper object (the bib table will be extracted)

Valid selects for this route are: abstract, URL, resource, member, posted, score, created, degree, update-policy, short-title, license, ISSN, container-title, issued, update-to, issue, prefix, approved, indexed, article-number, clinical-trial-number, accepted, author, group-title, DOI, is-referenced-by-count, updated-by, event, chair, standards-body, original-title, funder, translator, published, archive, published-print, alternative-id, subject, subtitle, published-online, publisher-location, content-domain, reference, title, link, type, publisher, volume, references-count, ISBN, issn-type, assertion, deposited, page, content-created, short-container-title, relation, editor

Value

doi

Examples

```
ref <- paste(
  "Lakens, D., Mesquida, C., Rasti, S., & Ditroilo, M. (2024).",
  "The benefits of preregistration and Registered Reports.",
  "Evidence-Based Toxicology, 2(1).",
)
## Not run:
cr <- crossref_query(ref)

## End(Not run)
```

datacite_doi

Doi.org Info from DataCite

Description

Doi.org Info from DataCite

Usage

```
datacite_doi(doi)
```

Arguments

doi the DOI(s) to get info for

Value

bib_match data frame

Examples

```
doi <- "10.5281/zenodo.2669586"
## Not run:
doi_info <- datacite_doi(doi)

## End(Not run)
```

dataone_info	<i>Retrieve info from DataONE by URL</i>
--------------	--

Description

Retrieve info from DataONE by URL

Usage

```
dataone_info(dataone_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

dataone_url	a DataONE URL or DOI, or a table containing them (e.g., as created by dataone_links())
id_col	the index or name of the column that contains DataONE URLs, if dataone_url is a table
pb	a progress bar passed from another function
cache	if TRUE, reuse a previously cached listing for a dataset already looked up (see repo_info_cache()) instead of re-querying the host's API. Off by default.

Value

a data frame of information

Examples

```
## Not run:
dataone_info("https://doi.org/10.18739/A2GT5FG86")

## End(Not run)
```

dataone_links	<i>Find DataONE Links in Papers</i>
---------------	-------------------------------------

Description

Get all DataONE links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE DOI mention (routinely cited without any URL scheme at all) using the DOI prefixes of known hosts (see [.dataone_hosts\(\)](#)) – the url table only ever contains links the source document itself made clickable. Same two-tier approach [zenodo_links\(\)](#) uses.

Usage

```
dataone_links(paper)
```

Arguments

paper a paper object or paperlist object

Details

Unlike the single-host archive modules, a DataONE citation only resolves to a specific member node's own API (see the file-level note in archive-dataone.R) – a host neither listed by domain nor identifiable by its DOI prefix is not one this package can currently query, even if it is a real DataONE member node.

Value

a table with the DataONE url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
dataone_links(psychsci)

## End(Not run)
```

dataverse_file_download

Download all files from a Dataverse dataset

Description

Creates a directory for the dataset and downloads all of its files. Returns (invisibly) a data frame with file info.

Usage

```
dataverse_file_download(
  host,
  doi,
  download_to = ".",
  max_file_size = 10,
  max_download_size = 100,
  unzip_types = NULL,
  pb = NULL
)
```

Arguments

host	the Dataverse installation's hostname (e.g. "dataverse.harvard.edu")
doi	the dataset's persistentId (a DOI, e.g. "10.7910/DVN/GGDUND")
download_to	path to download to
max_file_size	maximum file size to download (in MB) - set to NULL or Inf for no restrictions
max_download_size	maximum total size to download - set to NULL or Inf for no restrictions
unzip_types	file categories to extract from .zip files in the dataset rather than downloading the zip whole, named as <code>data_classify_files()</code> names them: "data", "code", "materials", "documentation", "output", "unknown". More than one may be given. NULL (the default) downloads zips whole and leaves them zipped, as before. <code>max_file_size</code> applies to each extracted member rather than to the archive.
pb	a progress bar passed from another function

Details

You can limit downloads to only files under a specific size (defaults to 10MB) and only a maximum download size (largest files will be omitted until total size is under the limit). Omitted files will be listed as messages in verbose mode, and included in the returned data frame with the downloaded column value set to FALSE.

A .zip in the dataset is normally fetched whole and left as a zip. Set `unzip_types` to pull only the files you want out of it instead: Dataverse's file-download endpoint honours byte-range requests (verified 2026-08-14 against demo.dataverse.org: HEAD/Range both return Accept-Ranges: bytes and a 206 Partial Content response) and each member of a zip is compressed separately, so the wanted members can be read out of the archive while the rest is never transferred – the same mechanism `zenodo_file_download()` uses. When the archive cannot be read that way, it is downloaded whole as before, so this never loses a file – it only avoids transferring one.

Value

data frame of file info. When members are extracted from a zip, its row reports the zip with extracted giving the number of members written; `size_on_disk` and `checksum_ok` are then NA, because what is on disk is the members rather than the archive Dataverse published a size and MD5 for.

Examples

```
## Not run:
dataverse_file_download("dataverse.harvard.edu", "10.7910/DVN/GGDUND")

# take only the data files out of any zip in the dataset
dataverse_file_download("dataverse.harvard.edu", "10.7910/DVN/GGDUND",
                        unzip_types = "data")

## End(Not run)
```

dataverse_info	<i>Retrieve info from Dataverse by URL</i>
----------------	--

Description

Retrieve info from Dataverse by URL

Usage

```
dataverse_info(dataverse_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

dataverse_url	a Dataverse dataset URL (or DOI, if host is also given), or a table containing them (e.g., as created by dataverse_links())
id_col	the index or name of the column that contains Dataverse URLs, if dataverse_url is a table
pb	a progress bar passed from another function
cache	if TRUE, reuse a previously cached listing for a dataset already looked up (see repo_info_cache()) instead of re-querying the installation's API. Off by default.

Value

a data frame of information

Examples

```
## Not run:
dataverse_info("https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/GGDUND")

## End(Not run)
```

dataverse_links	<i>Find Dataverse Links in Papers</i>
-----------------	---------------------------------------

Description

Get all Dataverse links: real hyperlinks from the paper's own url table, matched against a list of known Dataverse installations (see [dataverse_links\(\)](#)'s source for the list; Dataverse is open-source software run by many independent hosts, unlike Zenodo, so there is no single hostname or DOI prefix to match against everything). Same two-tier hyperlink + bare-mention approach [zenodo_links\(\)](#) uses, applied per known host.

Usage

```
dataverse_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the Dataverse url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
dataverse_links(psychsci)

## End(Not run)
```

dataverse_pat	<i>Set or get a Dataverse API token</i>
---------------	---

Description

Dataverse installations each issue their own API token (unlike Zenodo/OSF, which are single services) – a token from `dataverse.harvard.edu` is meaningless to `dataverse.nl`. Tokens are therefore stored per host.

Usage

```
dataverse_pat(host, pat = NULL)
```

Arguments

host the Dataverse installation's hostname (e.g. "dataverse.harvard.edu")
pat the token to set, or NULL to get the current token

Details

Create a token by signing in to the installation, opening your account page, and selecting "API Token".

Store it as an environment variable named `DATAVERSE_PAT_<HOST>`, with the host uppercased and every `.` replaced by `_` (e.g. `DATAVERSE_PAT_DATAVERSE_HARVARD_EDU`), so it is read every time R starts.

Value

the current token (character; "" when none is set)

Examples

```
dataverse_pat("dataverse.harvard.edu") # returns "" unless a token is set
```

`demofile`*Get a demo file*

Description

Return the file path for various versions of the demo paper. Use `demopaper()` to directly read it as a paper object from the json file.

Usage

```
demofile(ext = c("json", "pdf", "docx", "doc", "xml", "qmd"))
```

Arguments

`ext` the extension of the file

Value

file path

Examples

```
json <- demofile()  
pdf <- demofile("pdf")
```

`demopaper`*Get demo paper*

Description

Get demo paper

Usage

```
demopaper()
```

Value

paper object

Examples

```
paper <- demopaper()
```

`doi_clean`*Clean DOIs*

Description

Clean DOIs

Usage

```
doi_clean(doi)
```

Arguments

`doi` a character vector of one or more DOIs

Value

a character vector of cleaned DOIs (no `https://doi.org` or DOI:)

Examples

```
doi_clean("https://doi.org/10.1038/nphys1170")
doi_clean("doi:10.1038/nphys1170")
doi_clean("DOI: 10.1038/nphys1170")
```

`doi_lookup`*Doi.org Info from DOI*

Description

Doi.org Info from DOI

Usage

```
doi_lookup(doi)
```

Arguments

`doi` the DOI(s) to get info for

Value

data frame with DOIs and info

Examples

```
doi <- "10.7717/peerj.4375"
## Not run:
doi_info <- doi_lookup(doi)

## End(Not run)
```

doi_resolves

*Check whether a DOI resolves***Description**

Checks the doi.org API to see if a DOI is registered and has an associated URL (using <https://doi.org/api/handles>). Returns TRUE if it does, FALSE if the DOI does not exist or does not have an associated URL, and NA if the test failed. Clearly invalid DOIs (i.e. not starting with "10.") will return FALSE without server requests.

Usage

```
doi_resolves(doi, timeout = 10)
```

Arguments

doi	Character vector. One or more DOIs to check.
timeout	Numeric. Request timeout in seconds. Default is 10.

Value

Logical vector. For each input DOI, returns TRUE if the DOI resolves, FALSE if it does not resolve (or does not start with 10.), and NA if the check failed.

Examples

```
## Not run:
doi_resolves("10.1038/nphys1170") # Expected: TRUE
doi_resolves("10.1234/invalid.doi") # Expected: FALSE

## End(Not run)
```

doi_valid_format	<i>Validate DOI format</i>
------------------	----------------------------

Description

Validate DOI format

Usage

```
doi_valid_format(doi)
```

Arguments

doi a character vector of one or more DOIs

Value

a logical vector

Examples

```
doi_valid_format("10.1038/nphys1170")
doi_valid_format("no.no.10.1038")
```

dryad_auth	<i>Set or get Dryad OAuth2 client credentials</i>
------------	---

Description

Dryad also issues a client id/secret pair (from the same account settings page as the plain token in [dryad_pat\(\)](#)), meant for the OAuth2 client-credentials grant. Unlike a plain token, which expires after 10 hours and must be regenerated by hand, a client id/secret is exchanged for a short-lived access token automatically – and refreshed automatically, for as long as the credentials themselves stay valid – every time a request needs one, via `httr2::req_oauth_client_credentials()`. This is the right choice for a long-running corpus build; [dryad_pat\(\)](#) remains available for a single quick session where generating a client id/secret is not worth the extra step.

Usage

```
dryad_auth(
  client_id = NULL,
  client_secret = NULL,
  token_url = "https://datadryad.org/oauth/token"
)
```

Arguments

<code>client_id</code>	the OAuth2 client id, or NULL to get the current client
<code>client_secret</code>	the OAuth2 client secret, required when <code>client_id</code> is supplied
<code>token_url</code>	the token endpoint; Dryad's documented default should not normally need over-riding

Details

Store `client_id/client_secret` as the `DRYAD_CLIENT_ID/ DRYAD_CLIENT_SECRET` environment variables so they are read every time R starts, the same way `dryad_pat()` reads `DRYAD_PAT`.

Value

invisibly, the `httr2::oauth_client()` built from the current credentials, or NULL when none are set

Examples

```
## Not run:
dryad_auth("my-client-id", "my-client-secret")

## End(Not run)
```

<code>dryad_file_download</code>	<i>Download all files from a Dryad dataset</i>
----------------------------------	--

Description

Creates a directory for the dataset and downloads all of its files. Returns (invisibly) a data frame with file info.

Usage

```
dryad_file_download(
  dryad_doi,
  download_to = ".",
  max_file_size = 10,
  max_download_size = 100,
  unzip_types = NULL,
  pb = NULL
)
```

Arguments

<code>dryad_doi</code>	the dataset's DOI, or a URL containing it
<code>download_to</code>	path to download to
<code>max_file_size</code>	maximum file size to download (in MB) - set to NULL or Inf for no restrictions
<code>max_download_size</code>	maximum total size to download - set to NULL or Inf for no restrictions
<code>unzip_types</code>	file categories to extract from .zip files in the dataset rather than downloading the zip whole, named as <code>data_classify_files()</code> names them: "data", "code", "materials", "documentation", "output", "unknown". More than one may be given. NULL (the default) downloads zips whole and leaves them zipped, as before. <code>max_file_size</code> applies to each extracted member rather than to the archive.
<code>pb</code>	a progress bar passed from another function

Details

A Dryad API token (see `dryad_pat()`) is REQUIRED to download file bytes, even from a fully public dataset – unlike `figshare_file_download()`, `dataverse_file_download()`, or `zenodo_file_download()`. Without one set, every file in the returned table has `downloaded = FALSE`: the dataset can still be listed (see `dryad_info()`), but nothing can be fetched.

You can limit downloads to only files under a specific size (defaults to 10MB) and only a maximum download size (largest files will be omitted until total size is under the limit). Omitted files will be listed as messages in verbose mode, and included in the returned data frame with the `downloaded` column value set to FALSE.

A .zip in the dataset is normally fetched whole and left as a zip. Set `unzip_types` to pull only the files you want out of it instead, the same as `dataverse_file_download()` and `zenodo_file_download()` do.

Value

data frame of file info. When members are extracted from a zip, its row reports the zip with extracted giving the number of members written; `size_on_disk` and `checksum_ok` are then NA, because what is on disk is the members rather than the archive Dryad published a size and digest for.

Examples

```
## Not run:
dryad_file_download("10.5061/dryad.j1fd7")

# take only the data files out of any zip in the dataset
dryad_file_download("10.5061/dryad.j1fd7", unzip_types = "data")

## End(Not run)
```

dryad_info	<i>Retrieve info from Dryad by URL</i>
------------	--

Description

Retrieve info from Dryad by URL

Usage

```
dryad_info(dryad_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

dryad_url	a Dryad dataset URL or DOI, or a table containing them (e.g., as created by dryad_links())
id_col	the index or name of the column that contains Dryad URLs, if dryad_url is a table
pb	a progress bar passed from another function
cache	if TRUE, reuse a previously cached listing for a dataset already looked up (see repo_info_cache()) instead of re-querying Dryad's API. Off by default. Worth enabling for a corpus build likely to be interrupted and restarted, since Dryad's per-IP request quota is otherwise re-spent on every restart re-listing datasets nothing new needs fetching for.

Value

a data frame of information, one row per link, with the dataset's metadata, its file records in the list column files, and storage_size (the dataset's total size in bytes as Dryad reports it), files_total (the number of files Dryad reports), and files_incomplete (TRUE when fewer files could be listed than Dryad reports; a warning is given too). A sharing link (datadryad.org/stash/share/...) is followed to the dataset it leads to; one that leads nowhere has its reason in error, as a dataset that could not be found does.

Examples

```
## Not run:  
dryad_info("https://doi.org/10.5061/dryad.j1fd7")  
  
## End(Not run)
```

dryad_links

*Find Dryad Links in Papers***Description**

Get all Dryad links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE mention (a DOI like "10.5061/dryad.j1fd7" is routinely cited without any URL scheme at all) that the source PDF/HTML never encoded as an actual hyperlink – the url table only ever contains links the source document itself made clickable. Same two-tier approach zenodo_links() uses.

Usage

```
dryad_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the Dryad url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
dryad_links(psychsci)

## End(Not run)
```

dryad_pat

*Set or get a Dryad API token***Description**

Dryad issues OAuth2 bearer tokens from account settings. Unlike Figshare, Dataverse, Zenodo, and OSF, a token is REQUIRED to download file bytes even from a fully public dataset (verified live 2026-08-16: both the per-file and whole-dataset download endpoints answer 401 with no token, though dataset metadata and file listings are readable without one). Without a token, [dryad_file_download\(\)](#) can list what a dataset contains but cannot fetch any of it.

Usage

```
dryad_pat(pat = NULL)
```

Arguments

pat the token to set, or NULL to get the current token

Details

This token expires after 10 hours and must be regenerated by hand from Dryad's account settings – for a long corpus run, `dryad_auth()` avoids this by exchanging a client id/secret for a token automatically, with no manual re-entry. When client credentials are set via `dryad_auth()`, they take priority over this token.

Store it as the DRYAD_PAT environment variable so it is read every time R starts.

Value

the current token (character; "" when none is set)

Examples

```
dryad_pat() # returns "" unless a token is set
```

dspace_links	<i>Find DSpace (legacy REST API) Links in Papers</i>
--------------	--

Description

Get all links to any of the legacy-DSpace-REST installations in `.dspace_legacy_hosts()` (PsychArchives plus the others verified 2026-09-11 – see the note at the top of this file): real hyperlinks from the paper's own url table, plus a body-text fallback. The fallback recognises a bare mention (no hostname attached) only for PsychArchives' own handle prefix (20.500.12034) – see `.dspace_legacy_parse()` for why the other installations' bare mentions cannot be resolved the same way. This is the general (all-hosts) counterpart to `psycharchives_links()`, which stays PsychArchives-only for anyone calling it directly.

Usage

```
dspace_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the DSpace url in the first (href) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
dspace_links(psychsci)

## End(Not run)
```

`dspace7_file_download` *Retrieve public file list from a DSpace 7+ installation by URL*

Description

Lists the publicly retrievable files (the item's ORIGINAL bundle) of one or more DSpace 7 items via the REST API, without downloading them. Each row carries an absolute `file_url` so the actual bytes are fetched later by [download_repo_files](#), the same deferred path used for Zenodo, OSF, and legacy DSpace (`psycharchives_file_download()`).

Usage

```
dspace7_file_download(dspace7_url, pb = NULL)
```

Arguments

`dspace7_url` a vector of DSpace 7 item URLs
`pb` a progress bar passed from another function

Value

a data frame of file information (one row per public file)

`dspace7_links` *Find DSpace 7+ Links in Papers*

Description

Get all links to any of the DSpace 7 (or later) installations in `.dspace7_hosts()` – see the note at the top of this file for how they were found and why they need a separate implementation from `dspace_links()` (legacy DSpace, archive-psycharchives.R). Real hyperlinks from the paper's own `url` table, plus a body-text fallback for a bare mention that still carries one of these hosts (a bare handle prefix with no host at all cannot be attributed to a specific installation – see `.dspace7_parse()`).

Usage

```
dspace7_links(paper)
```

Arguments

paper a paper object or paperlist object

Details

A DOI mentioned in the text is also looked up, and kept when it leads to one of these installations. The paper's own DOI, and DOIs under a prefix that another detector handles and no DSpace 7 installation uses (see `.dspace7_skip_doi_prefixes()`), are not looked up. The lookup reads the URL registered for the DOI from doi.org's handle API rather than visiting the landing page, and each DOI is looked up once per R session.

Value

a table with the DSpace 7 url in the first (href) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
dspace7_links(psychsci)

## End(Not run)
```

email	<i>Set or get email</i>
-------	-------------------------

Description

Get or set the contact email metacheck sends with requests to external APIs (e.g. Crossref's polite pool), so a host that needs to reach you about your usage can do so. Call with no argument to read the current value; call with a valid email address to set it for the rest of the session. Defaults to "metacheck@scienceverse.org" when never set.

Usage

```
email(email = NULL)
```

Arguments

email if a string, sets the email

Value

the current option value (character)

Examples

```
email()
```

emojis	<i>Emojis</i>
Description	
Useful emojis	
Usage	
emojis	
Format	
An object of class list of length 32.	
Details	
• General: "check", "star", "warning", "stop", "x", "no", "thumbs_up", "thumbs_down", "info", "question" • Traffic Lights: "tl_green", "tl_yellow", "tl_red", "tl_info", "tl_na", "tl_fail" • Hearts: "red", "orange", "yellow", "green", "blue", "purple", "brown", "black", "white", "pink"	
export_jasp_html	<i>Export a JASP (.jasp) file's own rendered output as standalone HTML</i>
Description	
A .jasp archive already bundles a fully rendered index.html – JASP's own output view, complete with result tables and any plots – alongside the data (see the file header). This extracts that index.html as-is and inlines every plot it references (an) as a base64 data: URI, so the result is a single, portable, self-contained file that looks exactly like JASP's own output window, with no external image files to keep alongside it.	
Usage	
export_jasp_html(path, out = NULL)	
Arguments	
path	path to a .jasp file
out	path to write the HTML file to; defaults to path with its extension replaced by .html, written alongside the source file
Value	
the path written to, invisibly	

export_mplus_html	<i>Export an Mplus (.out) output file as standalone HTML</i>
-------------------	--

Description

Builds an HTML page from what `import_mplus_output()` already decodes: one heading per section (SUMMARY OF ANALYSIS, MODEL FIT INFORMATION, MODEL RESULTS, ...) and one `<table>` per detected result. Like `export_stata_smcl_html()`, this has no figures to embed: Mplus never writes chart data into the .out text file itself (see the file header).

Usage

```
export_mplus_html(path, out = NULL)
```

Arguments

path	path to a .out file
out	path to write the HTML file to; defaults to path with its extension replaced by .html, written alongside the source file

Value

the path written to, invisibly

export_omv_html	<i>Export a jamovi (.omv) file's own rendered output as standalone HTML</i>
-----------------	---

Description

A .omv archive already bundles a fully rendered index.html – jamovi's own output view, complete with result tables and any plots – alongside the data (see the file header). This extracts that index.html as-is and inlines every plot it references (an `` under one of the per-analysis resources/ folders) as a base64 data: URI, so the result is a single, portable, self-contained file that looks exactly like jamovi's own output window, with no external image files to keep alongside it.

Usage

```
export_omv_html(path, out = NULL)
```

Arguments

path	path to a .omv file
out	path to write the HTML file to; defaults to path with its extension replaced by .html, written alongside the source file

Value

the path written to, invisibly

export_spv_html	<i>Export an SPSS Viewer (.spv) file's tables and charts as standalone HTML</i>
-----------------	---

Description

Unlike .jasp/.omv, a .spv archive carries no rendered view of its own (see the file header) – so, unlike `export_jasp_html()` and `export_omv_html()`, this does not re-export an existing document. It builds a new HTML page from what `import_spv()` already decodes: one heading per analysis (its recovered SPSS syntax shown underneath, when recoverable), one `<table>` per result table, and – for a chart entry (`import_spv()`'s `is_chart = TRUE`) – a scatter plot rendered from its own decoded (x, y) points with any fitted trend lines SPSS itself computed drawn over it (see `.spv_decode_chart()`), embedded as a base64 PNG the same way `export_jasp_html()` inlines its own plots.

Usage

```
export_spv_html(path, out = NULL)
```

Arguments

path	path to a .spv file
out	path to write the HTML file to; defaults to path with its extension replaced by .html, written alongside the source file

Value

the path written to, invisibly

export_stata_smcl_html	<i>Export a Stata (.smcl) output log as standalone HTML</i>
------------------------	---

Description

Builds an HTML page from what `import_stata_smcl()` already decodes: one heading per Stata command (its exact syntax, since a .smcl command echo IS the syntax) and one `<table>` per detected result. Unlike `export_jasp_html()/export_omv_html()` (which re-export an already-rendered view) or `export_spv_html()` (which renders decoded charts as images), this has no figures to embed at all: a .smcl file never contains chart image data (see the file header) – a graph export command is shown as recovered command text only.

Usage

```
export_stata_smcl_html(path, out = NULL)
```

Arguments

`path` path to a .smcl file

`out` path to write the HTML file to; defaults to path with its extension replaced by .html, written alongside the source file

Value

the path written to, invisibly

extract_eq	<i>Extract Equations</i>
------------	--------------------------

Description

List all equations in the text, returning the matched text (e.g., 't(28) = 2.4', 'p = 0.04') and document location in a table. This is the canonical extractor for reported statistics and effect sizes; modules that need statistics should read from this table rather than re-scanning the text.

Usage

```
extract_eq(paper)
```

Arguments

`paper` a paper object or paperlist object

Details

This will catch most comparators like $=<>\sim$ and most versions of scientific notation like 5.0×10^{-2} or $5.0e-2$. If you find any formats that are not correctly handled by this function, please contact the author.

Value

a data frame with one row per equation and the columns `lhs` (the statistic name, e.g. "t", "F", "p"), `df` (parenthetical degrees of freedom such as "(28)" or "(2, 57)", otherwise NA), `comp` (the comparator, e.g. "="), `rhs` (the reported value as text), `grp_id` (groups equations in the same sentence), `text_id`, and `paper_id`.

Examples

```
paper <- demopaper()
equations <- extract_eq(paper)
```

extract_p_values	<i>Extract P-Values</i>
------------------	-------------------------

Description

List all p-values in the text, returning the matched text (e.g., 'p = 0.04') and document location in a table.

Usage

```
extract_p_values(paper)
```

Arguments

paper	a paper object or paperlist object
-------	------------------------------------

Details

Note that this will not catch p-values reported like "the p-value is 0.03" because that results in a ton of false positives when papers discuss p-value thresholds. If you need to detect text like that, use the `text_search()` function and a custom pattern.

This will catch most comparators like `=<>~` and most versions of scientific notation like `5.0 x 10^-2` or `5.0e-2`. If you find any formats that are not correctly handled by this function, please contact the author.

Value

a table

Examples

```
paper <- demopaper()
p_values <- extract_p_values(paper)
```

extract_tests	<i>Extract the statistical tests a paper reports</i>
---------------	--

Description

Builds the paper-side counterpart to the analysis output that `reproducibility_check` extracts: one row per reported TEST, with its components kept together and the sentence it was reported in retained, so a matched result can be traced back to the claim it supports.

Usage

```
extract_tests(paper)
```

Arguments

paper a paper object (uses its \$eq and \$text)

Details

This is a purpose-built alternative to `extract_eq()` for matching. It reuses `extract_eq()`'s parsing of name <op> value fragments, then does three things that parsing does not:

- **isolates separate tests** within one sentence. `extract_eq()`'s `grp_id` is the sentence number, so a sentence reporting two t-tests yields a single ten-component "test" that no analysis can produce; here a repeated anchor (a second t, F, r, ...) starts a new test;
- **keeps the sentence**, its `text_id`, `paragraph_id` and `section_id`, so a match links a `result_id` to the sentence making the claim;
- **filters non-statistics**, so a figure's height = 4 or a sample's range = 18-29 is not offered as evidence. A run of components with no recognised test statistic in it is dropped entirely.

Value

a data.frame, one row per reported test, with `paper_id`, `test_no` (sequential within the paper), `text_id`, `paragraph_id`, `section_id`, `sentence` (the reporting sentence), `anchor` (the test statistic the report is built around, e.g. "t"), `n_components`, `components` (a list-column of name/comp/value/df), and `reported` (the test rendered back as text, e.g. "t(23) = 3.77, p = .001, d = 0.77").

extract_urls

Extract URLs

Description

Get a table of URLs from a paper or paperlist. Matches urls that start with http or doi:

Usage

```
extract_urls(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table

Examples

```
paper <- demopaper()
urls <- extract_urls(paper)
```

fig_image_view	<i>View a figure image</i>
----------------	----------------------------

Description

View a figure image

Usage

```
fig_image_view(paper, figure_id = 1)
```

Arguments

paper	a paper object
figure_id	the id for the figure to show

Value

plots the figure

Examples

```
paper <- demopaper()
fig_image_view(paper, 1)
fig_image_view(paper, 2)
```

figshare_file_download	<i>Download all files from a Figshare article</i>
------------------------	---

Description

Creates a directory for the article and downloads all of its files. Returns (invisibly) a data frame with file info.

Usage

```
figshare_file_download(
  figshare_id,
  download_to = ".",
  max_file_size = 10,
  max_download_size = 100,
  unzip_types = NULL,
  host = "api.figshare.com",
  pb = NULL
)
```

Arguments

figshare_id	a Figshare article ID, URL, or DOI
download_to	path to download to
max_file_size	maximum file size to download (in MB) - set to NULL or Inf for no restrictions
max_download_size	maximum total size to download - set to NULL or Inf for no restrictions
unzip_types	file categories to extract from .zip files in the article rather than downloading the zip whole, named as <code>data_classify_files()</code> names them: "data", "code", "materials", "documentation", "output", "unknown". More than one may be given. NULL (the default) downloads zips whole and leaves them zipped, as before. <code>max_file_size</code> applies to each extracted member rather than to the archive.
host	the Figshare-compatible API host to query. Default "api.figshare.com"; researchdata4tu_file_download calls this with "data.4tu.nl" instead (see the note at the top of this file).
pb	a progress bar passed from another function

Details

Files are downloaded one by one, each from its own `download_url`. Figshare does document a whole-article zip (GET /v2/articles/{id}/download, which redirects to `ndownloader.figshare.com/articles/<id>/v2/articles/{id}/download` and serves `application/zip` with a `Content-Length`, verified 2026-10-13), but it is not used here: it always contains every file, so it cannot honour the size limits below, and how it lays out files kept in folders was not verified. Files in folders (Figshare's `folder_structure`) are saved under those folders. A .zip already present in the article is handled the same way `dataverse_file_download()` handles one: fetched whole and left as a zip unless `unzip_types` asks for only specific members out of it.

You can limit downloads to only files under a specific size (defaults to 10MB) and only a maximum download size (largest files will be omitted until total size is under the limit). Omitted files will be listed as messages in verbose mode, and included in the returned data frame with the downloaded column value set to FALSE.

Value

data frame of file info. When members are extracted from a zip, its row reports the zip with extracted giving the number of members written; `size_on_disk` and `checksum_ok` are then NA, because what is on disk is the members rather than the archive Figshare published a size and MD5 for.

Examples

```
## Not run:
figshare_file_download("18093368")

# take only the data files out of any zip in the article
figshare_file_download("18093368", unzip_types = "data")

## End(Not run)
```

figshare_info	<i>Retrieve info from Figshare by URL</i>
---------------	---

Description

Retrieve info from Figshare by URL

Usage

```
figshare_info(
  figshare_url,
  id_col = 1,
  host = "api.figshare.com",
  pb = NULL,
  cache = FALSE
)
```

Arguments

figshare_url	a Figshare URL or DOI, or a table containing them (e.g., as created by figshare_links())
id_col	the index or name of the column that contains Figshare URLs, if figshare_url is a table
host	the Figshare-compatible API host to query. Default "api.figshare.com"; researchdata4tu_info() calls this with "data.4tu.nl" instead (see the note at the top of this file).
pb	a progress bar passed from another function
cache	if TRUE, reuse a previously cached listing for an article already looked up (see repo_info_cache()) instead of re-querying the API. Off by default.

Value

a data frame of information

Examples

```
## Not run:
  figshare_info("https://doi.org/10.6084/m9.figshare.18093368.v1")

## End(Not run)
```

figshare_links

*Find Figshare Links in Papers***Description**

Get all Figshare links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE mention (a DOI like "10.6084/m9.figshare.1234567" is routinely cited without any URL scheme at all) that the source PDF/HTML never encoded as an actual hyperlink – the url table only ever contains links the source document itself made clickable. Same two-tier approach zenodo_links() uses.

Usage

```
figshare_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the Figshare url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
figshare_links(psychsci)

## End(Not run)
```

figshare_pat

*Set or get a Figshare API token***Description**

Figshare issues personal access tokens from account settings (<https://figshare.com/account/applications>). A token is optional for reading public articles and is used here only to read private ones (it does not raise any request limit: Figshare documents no automatic rate limiting and asks for at most 1 request per second either way), unlike Dataverse where a token is per-installation.

Usage

```
figshare_pat(pat = NULL)
```

Arguments

pat the token to set, or NULL to get the current token

Details

Store it as the FIGSHARE_PAT environment variable so it is read every time R starts.

Value

the current token (character; "" when none is set)

Examples

```
figshare_pat() # returns "" unless a token is set
```

file_category	<i>Categorise files</i>
---------------	-------------------------

Description

Categorise files

Usage

```
file_category(contents)
```

Arguments

contents a table with columns name, path such as from osf_contents()

Value

the table with new column file_category

Examples

```
contents <- c("script.R", "data.csv", "README", "codebook.csv")
file_category(contents)
```

filetype	<i>Get file Type from Extension</i>
----------	-------------------------------------

Description

Get file Type from Extension

Usage

```
filetype(filename)
```

Arguments

filename the file name

Value

a named vector of file types

Examples

```
filetype("script.R")
```

FLoRA	<i>FORRT Replication Database (FLoRA)</i>
-------	---

Description

FLoRA database containing DOIs of original studies and replications. Use FLoRA_date() to find the date it was downloaded, and FLoRA_update() to update it.

Usage

```
FLoRA()
```

Format

A data frame with 8 columns:

doi_o DOI of original study

apa_ref_o APA reference of original study

doi_r DOI of replication study (may be NA if url_r is provided)

apa_ref_r APA reference of replication study

url_r URL of replication study (used when DOI is not available)

outcome replication outcome

outcome_quote quote describing replication outcome

type replication or reproduction

Value

a data frame

Source

<https://osf.io/9r62x/files/t4j8f>

Examples

```
FLoRA()
```

FLoRA_date	<i>Get date FLoRA was updated</i>
------------	-----------------------------------

Description

Get date FLoRA was updated

Usage

```
FLoRA_date()
```

Value

the date

Examples

```
FLoRA_date()
```

FLoRA_update	<i>Update FLoRA</i>
--------------	---------------------

Description

metacheck comes with a built-in data frame called FLoRA. We update it regularly, but you can use this function to download the newest version. The download is >5MB, but this function will summarise the information into a smaller version and delete the original file.

Usage

```
FLoRA_update()
```

Value

the path to the data frame (invisibly)

format_bib_authors	<i>Format Bib Authors</i>
--------------------	---------------------------

Description

Formats a structured author list (data frame with given/family columns) as a display string.

Usage

```
format_bib_authors(authors)
```

Arguments

authors a data frame with given and family columns, or a list of such data frames

Value

a character string (or vector) of formatted author names

Examples

```
authors <- data.frame(given = c("Alice H.", "Wendy"),
                      family = c("Eagly", "Wood"))
format_bib_authors(authors)
```

format_ref	<i>Format Reference</i>
------------	-------------------------

Description

Format a reference for display in a report.

Usage

```
format_ref(bib)
```

Arguments

bib a bibentry object or list of bibentry objects

Details

The argument bib should be a bibentry object (e.g., like those made by citation()), but it can also handle a bibtex object or a bibtex formatted character vector. If these do not read in as valid bibtex, the original text of bib will be returned unformatted.

Value

formatted text

Examples

```
mc <- citation("metacheck")
format_ref(mc)

# handles bibtext
bib_mc <- utils::toBibtex(mc)
format_ref(bib_mc)

paper <- demopaper()
format_ref(paper$bib$ref[1:2])
```

fsd_info

Retrieve info from FSD by URL

Description

Lists a Finnish Social Science Data Archive study's title, DOI, access restriction, and data-file inventory (name, format, case/variable counts) via its open DDI-C 2.5 XML metadata record. There is no file-level download here – the DDI record carries no download URI, and actual data access requires Haka/Aila login or a per-dataset access application (see the file header comment) – so files is for REPORTING what a study contains, not for `download_repo_files()` to fetch.

Usage

```
fsd_info(fsd_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

<code>fsd_url</code>	an FSD URL (catalogue page, DOI, or URN), or a table containing them (e.g., as created by <code>fsd_links()</code>)
<code>id_col</code>	the index or name of the column that contains FSD URLs, if <code>fsd_url</code> is a table
<code>pb</code>	a progress bar passed from another function
<code>cache</code>	if TRUE, reuse a cached listing from a prior call for the same study instead of re-querying FSD (see <code>repo_info_cache()</code>); default FALSE

Value

a data frame of information

Examples

```
## Not run:
fsd_info("https://services.fsd.tuni.fi/catalogue/FSD2653")

## End(Not run)
```

fsd_links*Find FSD Links in Papers*

Description

Get all Finnish Social Science Data Archive (FSD / Aila Data Service) links: real hyperlinks from the paper's own url table (the catalogue page, a 10.60686 DOI, or a urn.fi URN), plus a body-text fallback for a bare "FSD#####" mention that the source PDF/HTML never encoded as an actual hyperlink. Same two-tier approach `github_links()` uses for GitHub.

Usage

```
fsd_links(paper)
```

Arguments

`paper` a paper object or paperlist object

Value

a table with the FSD url in the first (href) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
fsd_links(psychsci)

## End(Not run)
```

get_prev_outputs*Get Previous Outputs*

Description

A helper for creating modules. Checks for previous module outputs in a chain and returns the named list item if it exists in any parent environment.

Usage

```
get_prev_outputs(module, item, parent_n = 2)
```

Arguments

module	the name of a previously run module
item	the name of the list item to extract
parent_n	the number of parents to traverse up the chain. Normally 2 if you are calling this from a module function, but maybe more if you are calling it from a helper function.

Value

the extracted list item, or NULL if not found

Examples

```
# .__mc__prev_outputs is usually created by `module_run()`
.__mc__prev_outputs <- list(mod_1 = list(a = 1, b = 2))
f <- function(item) {
  get_prev_outputs("mod_1", item)
}
f("a")
f("d")
```

github_files

Get File List from GitHub

Description

Get File List from GitHub

Usage

```
github_files(repo, dir = "", recursive = FALSE)
```

Arguments

repo	The URL of the repository (in the format "username/repo" or "https://github.com/username/repo")
dir	an optional directory name to search
recursive	whether to search the files recursively

Value

a data frame of files

Examples

```
## Not run:
github_files("scienceverse/metacheck")

## End(Not run)
```

github_info	<i>Get GitHub Repo Info</i>
-------------	-----------------------------

Description

Get GitHub Repo Info

Usage

```
github_info(repo, recursive = FALSE)
```

Arguments

repo	The URL of the repository (in the format "username/repo" or "https://github.com/username/repo")
recursive	whether to search the files recursively

Value

a list of information about the repo

Examples

```
## Not run:  
github_info("scienceverse/metacheck")  
  
## End(Not run)
```

github_languages	<i>Get Languages from GitHub Repo</i>
------------------	---------------------------------------

Description

Get Languages from GitHub Repo

Usage

```
github_languages(repo)
```

Arguments

repo	The URL of the repository (in the format "username/repo" or "https://github.com/username/repo")
------	---

Value

vector of languages

Examples

```
## Not run:
github_languages("scienceverse/metacheck")

## End(Not run)
```

github_links	<i>Find GitHub Links in Papers</i>
--------------	------------------------------------

Description

GitHub links can be in PDFs in several ways.

Usage

```
github_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the GitHub url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
github_links(psychsci)

## End(Not run)
```

github_readme	<i>Get README from GitHub</i>
---------------	-------------------------------

Description

Get README from GitHub

Usage

```
github_readme(repo)
```

Arguments

repo The URL of the repository (in the format "username/repo" or "https://github.com/username/repo")

Value

a character string of the README contents

Examples

```
## Not run:  
github_readme("scienceverse/metacheck")  
  
## End(Not run)
```

github_repo	<i>Get Short GitHub Repo Name</i>
-------------	-----------------------------------

Description

Get Short GitHub Repo Name

Usage

```
github_repo(repo)
```

Arguments

repo The URL of the repository (in the format "username/repo" or "https://github.com/username/repo")

Value

character string of short repo name

Examples

```
github_repo("scienceverse/metacheck")  
github_repo("https://github.com/scienceverse/metacheck/")  
github_repo("https://github.com/scienceverse/metacheck.git")
```

gitlab_links	<i>Find GitLab Links in Papers</i>
--------------	------------------------------------

Description

GitLab.com links can appear in papers the same two ways GitHub links do: a real hyperlink, or a bare "owner/repo" mention near the word "gitlab". Only gitlab.com is matched; a self-hosted GitLab instance's own domain is indistinguishable from an arbitrary institutional website (see the file header comment).

Usage

```
gitlab_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the GitLab url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
gitlab_links(psychsci)

## End(Not run)
```

gitlab_pat	<i>Set or get a GitLab API token</i>
------------	--------------------------------------

Description

A personal access token from gitlab.com account settings. Optional for listing/downloading PUBLIC projects (used here only to raise GitLab's unauthenticated rate limit); required for private projects.

Usage

```
gitlab_pat(pat = NULL)
```

Arguments

pat the token to set, or NULL to get the current token

Details

Store it as the GITLAB_PAT environment variable so it is read every time R starts.

Value

the current token (character; "" when none is set)

Examples

```
gitlab_pat() # returns "" unless a token is set
```

gitlab_repo	<i>Get Short GitLab Project Path</i>
-------------	--------------------------------------

Description

Get Short GitLab Project Path

Usage

```
gitlab_repo(repo)
```

Arguments

repo	The URL of the project (in the format "namespace/project" or "https://gitlab.com/namespace/project")
------	--

Value

character string of the "namespace/project" path, or NULL if the project does not exist / is not reachable

Examples

```
gitlab_repo("gitlab-org/gitlab")
gitlab_repo("https://gitlab.com/gitlab-org/gitlab/")
gitlab_repo("https://gitlab.com/gitlab-org/gitlab.git")
```

grobid_to_bibr	<i>Convert Grobid TEI XML file to bibr format</i>
----------------	---

Description

Convert Grobid TEI XML file to bibr format

Usage

```
grobid_to_bibr(
  xml_path,
  save_path = ".",
  crossref_lookup = FALSE,
  schema_version = NULL
)
```

Arguments

xml_path	the path to the XML file
save_path	directory or file path to save to; set to NULL to return a paper object
crossref_lookup	whether to look up references in crossref
schema_version	NULL (the default) converts as before; "12.0" makes a paper in bibr export schema 12.x form and saves a bibr 12.0 file (see paper_write()). Its source is the PDF Grobid read when it is next to the XML (published.pdf for published.pdf.tei.xml or published.xml), else the XML file.

Value

a paper object

import_jasp	<i>Read a JASP (.jasp) file</i>
-------------	---------------------------------

Description

Extracts the dataset, its variable metadata (measurement level, value labels) and the list of analyses stored in a .jasp archive. Handles both the legacy binary format and the modern embedded-SQLite format.

Usage

```
import_jasp(path)
```

Arguments

path path to a .jasp file

Value

a list with data (a data.frame; labelled columns carry haven-style label/labels attributes), columns (a data.frame of name and type), analyses (the parsed analyses.json, or NULL), format ("binary" or "sqlite"), and data_file_path (the original source path recorded in the archive, or NA).

import_mplus_output	<i>Read an Mplus (.out) output file</i>
---------------------	---

Description

Splits the file into its ALL-CAPS-headed sections (Mplus's own structural marker – see the file header) and extracts each section's fixed-width result tables and label/value statistics. The INPUT INSTRUCTIONS section (Mplus's own verbatim analysis syntax) is skipped here – see [.mplus_export_syntax\(\)](#) for recovering it separately as a sibling .inp file, the same way [.spv_export_syntax\(\)](#) recovers .sps from .spv.

Usage

```
import_mplus_output(path)
```

Arguments

path path to a .out file

Value

a list of result tables, each list(analysis, title, data, syntax, table_index) – the same shape [read_stat_tables\(\)](#) returns for .jasp/.omv/.spv, so all formats can be processed identically downstream. analysis is the section title (e.g. "MODEL RESULTS"); syntax is the file's own recovered INPUT INSTRUCTIONS block (the same text on every table, since one .out file is one Mplus run). Empty list if the file has no recoverable section.

import_omv	<i>Read a jamovi (.omv) file</i>
------------	----------------------------------

Description

Extracts the dataset, its variable metadata (measurement level, value labels) and a summary of the analyses stored in a .omv archive. The jamovi counterpart of `import_jasp()`, returning the same structure so downstream code (codebook extraction, data checks) treats an .omv like a .jasp or .sav.

Usage

```
import_omv(path)
```

Arguments

path	path to a .omv file
------	---------------------

Value

a list with data (a data.frame; labelled columns carry haven-style label/labels attributes), columns (a data.frame of name and type), analyses (a character vector, one entry per analysis: its name and, when recoverable, the reproducible R syntax), format ("jamovi"), and data_file_path (NA; jamovi does not record the original import path).

import_spv	<i>Read the statistical result tables from an SPSS Viewer (.spv) file</i>
------------	---

Description

Ties the structure reader (`.spv_read_structure()`, which table exists, what analysis produced it, which format its data is in) and the two decoders (`.spv_decode_light_table()` for modern tables, `.spv_decode_legacy_data()` + `.spv_decode_legacy_table()` for pre-21 tables) into the SAME shape `read_stat_tables()` already returns from JASP's analyses.json and jamovi's protobuf blobs.

Usage

```
import_spv(path)
```

Arguments

path	path to a .spv file
------	---------------------

Details

A table that fails to decode (an unsupported construct in either format, or real malformation) is skipped rather than aborting the whole file, so one bad table does not lose every other result in the same archive.

Value

a list of result tables, each `list(analysis, title, data, syntax, table_index)` — the same shape `read_stat_tables()` returns for `.jasp/.omv` (syntax is `.spv`-specific: the exact SPSS syntax that produced the table, when recoverable). Empty list if the archive has no structure XML or no tables decode.

<code>import_stata_smcl</code>	<i>Read a Stata Markup and Control Language (.smcl) output log</i>
--------------------------------	--

Description

Renders the file's markup to plain text (see `.smcl_render()`), splits it into one chunk per Stata command (the `{com}`. `<command>` echo, mirroring `read_r_output()`'s R-console-echo splitting), and extracts each chunk's result tables and one-line statistics. `graph export "...png"` commands are recovered as ordinary command text; the PNG itself is never embedded in a `.smcl` file (Stata writes it to disk separately), so no chart data exists in this format to extract (see the file header).

Usage

```
import_stata_smcl(path)
```

Arguments

`path` path to a `.smcl` file

Value

a list of result tables, each `list(analysis, title, data, syntax, table_index)` — the same shape `read_stat_tables()` returns for `.jasp/.omv/.spv`, so all four formats can be processed identically downstream. `analysis` is the Stata command that produced the table (e.g. `"summarize communication_index barrier_index"`); `syntax` duplicates `analysis` (Stata's own echoed command IS its syntax, unlike `.spv` where syntax is recovered separately from a different structure). Empty list if the file has no recoverable command output.

 json_expand

Expand a JSON column

Description

It is useful to ask an LLM to return data in JSON structured format, but can be frustrating to extract the data, especially where the LLM makes syntax mistakes. This function tries to expand a column with a JSON-formatted response into columns and deals with it gracefully (sets an 'error' column to "parsing error") if there are errors. It also fixes column data types, if possible.

Usage

```
json_expand(table, col = "answer", suffix = c("", ".json"))
```

Arguments

table	the table with a column to expand
col	the name or index of the column to expand (defaults to "answer" or the first column)
suffix	the suffix for the extracted columns if they conflict with names in the table

Value

the table plus the expanded columns

Examples

```
table <- data.frame(
  paper_id = 1:5,
  answer = c(
    '{"number": "1", "letter": "A", "bool": true}',
    '{"number": "2", "letter": "B", "bool": "FALSE"}',
    '{"number": "3", "letter": "", "bool": null}',
    "oh no, the LLM misunderstood",
    '{"number": "5", "letter": ["E", "F"], "bool": false}'
  )
)

expanded <- json_expand(table, "answer")
expanded
```

lastlog	<i>Get the last log</i>
---------	-------------------------

Description

Reads entries back from the on-disk log written by `logger()` (newest first, so `i = 1` is the most recent entry). Returns a single entry as a list, or several entries row-bound into a data frame.

Usage

```
lastlog(i = 1, logpath = NULL)
```

Arguments

<code>i</code>	the indices to return
<code>logpath</code>	an optional file path to read the log from

Value

a list of the last log item, or a data frame of multiple items

Examples

```
# set up 2 log items
logger("test", list(msg = "hi"))
logger("test", list(msg = "hi again"))

lastlog()
lastlog(2)
lastlog(1:2)
```

ldbase_info	<i>Get the files of LDbase projects, datasets, code and documents</i>
-------------	---

Description

Lists the files of each LDbase link: for a project, every dataset, code and document in its hierarchy (child projects included); for a single dataset, code or document, its own file. A DOI is first resolved to the page it leads to. LDbase has no API, so this reads the public pages (one request per project or item). Items stored elsewhere (LDbase's "external link" option) have no file here and are not listed.

Usage

```
ldbase_info(ldbase_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

ldbase_url	a vector of LDbase URLs or DOIs, or a table containing them (e.g. from <code>ldbase_links()</code>)
id_col	the index or name of the column that contains the URLs, if ldbase_url is a table
pb	a progress bar from <code>pb()</code> , or NULL to create one
cache	if TRUE, a listing is read from / written to the persistent repository-listing cache (see <code>repo_info_cache_clear()</code>)

Value

a data frame with one row per link: ldbase_url, ldbase_page (the resolved page), ldbase_type, title, doi, license (the licence of the item, or every distinct licence of a project's items joined by "; "), a list-column files (a data frame per link with file_name, file_path, file_url, item_type, item_url) and error (NA, or why the link could not be listed)

ldbase_links	<i>Find LDbase links in a paper</i>
--------------	-------------------------------------

Description

Finds links to LDbase (<https://ldbase.org>) projects, datasets, code and documents, as hyperlinks or in the text, and LDbase DOIs (prefix 10.33009, e.g. 10.33009/ldbase.1627053448.d151). LDbase's own DOI (10.33009/ldbase, cited for the repository itself) and links to its home or search pages are not included: they name no deposit.

Usage

```
ldbase_links(paper)
```

Arguments

paper	a paper object or paperlist object
-------	------------------------------------

Value

a data frame with a row for each link (href, plus text_id / paper_id), with ldbase_url (the link) and ldbase_type (projects, datasets, code, documents, or doi for a DOI not yet resolved)

Examples

```
paper <- test_paper(url = "https://ldbase.org/projects/09609d82-c45f-4442-8c0c-d136191bdc50")
ldbase_links(paper)
```

link	Make an html link								
Description Make an html link									
Usage <pre>link(url, text = url, new_window = TRUE, type = "")</pre>									
Arguments <table><tr><td>url</td><td>the URL to link to</td></tr><tr><td>text</td><td>the text to link</td></tr><tr><td>new_window</td><td>whether to open in a new window</td></tr><tr><td>type</td><td>handle common links, like "doi" ()</td></tr></table>		url	the URL to link to	text	the text to link	new_window	whether to open in a new window	type	handle common links, like "doi" ()
url	the URL to link to								
text	the text to link								
new_window	whether to open in a new window								
type	handle common links, like "doi" ()								
Value string									
Examples <pre>link("https://scienceverse.org")</pre>									
llm	Query an LLM								

Description Ask a large language model (LLM) any question you want about a vector of text or the text from a text_search(). When type is provided, uses ellmer's structured output API to guarantee output conforming to the type spec; otherwise returns free-text responses in an answer column.	
Usage <pre>llm(text, system_prompt, type = NULL, text_col = "text", model = llm_model(), params = list(), phase = NULL, capture_reasoning = FALSE)</pre>	

Arguments

<code>text</code>	The text to send to the LLM (vector of strings, or data frame with the text in a column)
<code>system_prompt</code>	A system prompt to set the behavior of the assistant
<code>type</code>	An optional ellmer type specification for structured extraction (e.g., from <code>type_object()</code> , <code>type_from_schema()</code>). When provided, the provider enforces the schema and returns structured columns instead of free text.
<code>text_col</code>	The name of the text column if <code>text</code> is a data frame
<code>model</code>	the LLM model name (see <code>llm_model_list()</code>) in the format "provider" or "provider/model"
<code>params</code>	a named list to pass to <code>ellmer::params()</code>
<code>phase</code>	optional label naming the calling step (e.g. "Identifying scales") shown in the progress bar so a slow LLM pass is identifiable; the model name is appended automatically
<code>capture_reasoning</code>	if TRUE (default FALSE), a structured call (type provided) adds a <code>.reasoning</code> column: the provider's own chain-of-thought text for that row (via ellmer's <code>ContentThinking</code> , e.g. from <code>reasoning_effort/think</code>), or NA when the provider/call returned none. Off by default since most callers never look at it and it costs nothing extra to skip; every existing caller is unaffected.

Details

You will need to get your own API key from <https://console.groq.com/keys>. To avoid having to type it out, add it to the `.Renviro`n file in the following format (you can use `usethis::edit_r_environ()` to access the `.Renviro`n file)

```
GROQ_API_KEY="key_value_asdf"
```

See <https://console.groq.com/docs> for more information

Value

a data frame of results

Examples

```
## Not run:
# Free-text query
text <- c("hello", "number", "ten", 12)
system_prompt <- "Is this a number? Answer only 'TRUE' or 'FALSE'"
is_number <- llm(text, system_prompt)

# Structured extraction
type_spec <- ellmer::type_object(
  is_number = ellmer::type_boolean("Whether the input is a number")
)
result <- llm(c("hello", "42"), "Classify the input.", type = type_spec)

## End(Not run)
```

llm_cache	<i>Enable, disable, or query the LLM response cache</i>
-----------	---

Description

When enabled (the default), `llm()` stores each structured or free-text response on disk and replays it on identical later calls, so re-running a report on the same paper does not re-issue (or re-bill) the LLM requests. Because `llm()` runs at temperature 0 the replayed answer matches a fresh call. Cached entries are keyed by model, system prompt, input text, type spec, and params; changing any of these produces a fresh call.

Usage

```
llm_cache(enabled = NULL)
```

Arguments

enabled	if logical, sets whether the cache is used; if NULL (default) returns the current setting
---------	---

Value

the current setting (logical), invisibly when setting

Examples

```
llm_cache()      # is the cache on?
## Not run:
llm_cache(FALSE) # force fresh calls

## End(Not run)
```

llm_cache_clear	<i>Clear the on-disk LLM response cache</i>
-----------------	---

Description

Deletes all cached LLM responses (see `llm_cache()`).

Usage

```
llm_cache_clear()
```

Value

the number of cache entries removed, invisibly

llm_max_calls	<i>Set the maximum number of calls to the LLM</i>
---------------	---

Description

Set the maximum number of calls to the LLM

Usage

```
llm_max_calls(n = NULL)
```

Arguments

n	The maximum number of calls that the llm() function can make
---	--

llm_max_tokens	<i>Set the default max_tokens for LLM calls</i>
----------------	---

Description

llm() needs SOME max_tokens value on every call (a provider truncates a response with no clear error otherwise – a codebook/data-check pass over many columns at once is the case most likely to need more than the built-in default). Rather than requiring every module call to repeat `params = list(max_tokens = ...)` individually, set it once per session here; llm() falls back to this whenever a caller's own params does not already specify max_tokens (an explicit `params$max_tokens` on a single call still wins, same precedence as every other params entry).

Usage

```
llm_max_tokens(n = NULL)
```

Arguments

n	the default max_tokens value, or NULL to read the current value without changing it
---	---

Value

the current default max_tokens value (invisibly when setting)

Examples

```
## Not run:
llm_max_tokens(8192)

## End(Not run)
```

llm_model	<i>Set the default LLM model</i>
-----------	----------------------------------

Description

Use llm_model_list() to get a list of available models

Usage

```
llm_model(model = NULL)
```

Arguments

model	the name of the model
-------	-----------------------

llm_model_list	<i>List LLM Models</i>
----------------	------------------------

Description

List available LLM models for the specified platform.

Usage

```
llm_model_list(platform = NULL)
```

Arguments

platform	The platform. If NULL, checks all platforms for which you have a valid API_KEY.
----------	---

Details

For platforms other than groq, returns the value from the corresponding ellmer::models_platform function.

Value

a data frame of models and info

Examples

```
## Not run:
llm_model_list()

## End(Not run)
```

llm_reasoning	<i>Set the default reasoning effort for LLM calls</i>
---------------	---

Description

Every LLM call this package makes is a bounded, structured-extraction task over text already given in the prompt (classify a file, extract a variable's label, list the scales named in ~30 sentences, group files by study) – pattern-matching over given material, not multi-step reasoning (no arithmetic, no multi-hop inference, no planning). A "thinking"/ "reasoning" model spending extra tokens re-deriving what the prompt already states is close to pure overhead here: it inflates cost/latency and eats into the same `max_tokens` budget as the visible answer, which is a confirmed real cause of truncated structured responses on gpt-oss models via Groq. Minimal reasoning effort is the right DEFAULT for what this package does, not a tradeoff against quality.

Usage

```
llm_reasoning(effort = NULL)
```

Arguments

<code>effort</code>	one of "low", "medium", "high", "none", or NULL to read the current value without changing it. Default recommendation for this package's own calls is "low" (or "none" where the model family supports fully disabling reasoning).
---------------------	--

Details

There is no single universal "reasoning" switch across providers – this is genuinely per model family, each verified rather than assumed:

- `openai/gpt-oss-*` (Groq or any OpenAI-compatible host): the `reasoning_effort` chat-completion parameter, accepting "low", "medium", or "high" – "low" is the minimum; there is no full-off option for this family (verified against Groq's own docs).
- `qwen3*-family` models: also `reasoning_effort`, but accepting "none" or "default" – "none" fully disables reasoning tokens, unlike gpt-oss.
- Mistral's reasoning-capable models (`mistral-small-latest`, `mistral-medium-3-5`): also `reasoning_effort`, but with only TWO values, "high" (full thinking chunk) or "none" (thinking chunk omitted) – verified against Mistral's own docs, a third distinct shape from both gpt-oss and qwen3.
- Ollama-hosted reasoning models (DeepSeek-R1, Qwen3, ...): the native API's boolean `think` (`llm()` already sends this as a `/nothink` system-prompt prefix over Ollama's native `/api/chat` endpoint when `think = FALSE`, since Ollama's OpenAI-compatible `/v1/` endpoint ignores it) – a genuinely different mechanism (a flag on the request, not a graded effort level), so `llm_reasoning()`'s "low"/ "none" both map to `think = FALSE` for this family; there is no Ollama equivalent of "medium"/"high" to request.

A model family with none of the above (a plain non-reasoning model, or one not yet covered here) is left untouched: `llm_reasoning()` never forces an unsupported parameter onto a call, since providers vary in whether an unrecognised parameter is silently ignored or rejected outright.

Value

the current default reasoning effort (invisibly when setting)

Examples

```
## Not run:  
llm_reasoning("low")  
  
## End(Not run)
```

`llm_timeout`*Set the per-call timeout for LLM requests*

Description

Every single request `llm()` makes (one item, or one batch, depending on the caller) is bounded by this wall-clock limit. Without it, a stuck local model (most often Ollama under load, or a batch too large for the model to finish) blocks silently – no log output, no progress, nothing – for as long as the underlying HTTP client is willing to wait, which can be hours. A timeout turns that invisible stall into a fast, visible, per-row failure (recorded the same way a provider error already is – see `llm()`'s own error handling), so a large batch run fails fast and retries instead of hanging.

Usage

```
llm_timeout(seconds = NULL)
```

Arguments

`seconds` the timeout in seconds, or `NULL` to read the current value without changing it

Value

the current timeout in seconds (invisibly when setting)

Examples

```
## Not run:  
llm_timeout(120)  
  
## End(Not run)
```

llm_use	<i>Set or get metacheck LLM use</i>
---------	-------------------------------------

Description

Mainly for use in optional LLM workflows in modules

Usage

```
llm_use(llm_use = NULL)
```

Arguments

llm_use if logical, sets whether to use LLMs

Value

the current option value (logical)

Examples

```
if (llm_use()) {  
  print("We can use LLMs")  
} else {  
  print("We will not use LLMs")  
}
```

local_files	<i>List Local Files</i>
-------------	-------------------------

Description

Lists all files in a local directory recursively and returns a data frame compatible with the repo_check output table, for use with code_check.

Usage

```
local_files(path, recursive = FALSE)
```

Arguments

path path to a local directory or file, or a vector of paths
recursive whether to search the files recursively

Value

a data frame with columns repo_url, file_name, file_url, file_location, file_size, file_type

Examples

```
## Not run:
  local_files("my_project")

## End(Not run)
```

 logger

Log messages

Description

Adds a logging message to the log. Keeps the log as a maximum of 1000 rows.

Usage

```
logger(label = "", contents = list(), logpath = NULL)
```

Arguments

label	a string with the context (e.g., module name)
contents	a named list of the log contents
logpath	an optional file path to save the log in

Value

called for side effects of writing to log, returns logpath

Examples

```
logpath <- tempfile(fileext = ".log")
logger("test", list(x = 1), logpath)
lastlog()
```

 manifest_merge

Merge fields into a metacheck manifest, preserving other sections

Description

The per-paper *.manifest.json is written by more than one module: data_check records the files/provenance, and code_check records the packages the code loads (a code section). Because each module rebuilds only its own part of the document, a plain overwrite would let whichever module ran last erase the other's section. This helper reads any existing manifest, overlays patch at the top level (each key in patch replaces that key wholesale; keys not in patch are kept untouched), and writes it back.

Usage

```
manifest_merge(path, patch)
```

Arguments

path	the manifest file path (*.manifest.json)
patch	a named list of top-level keys to set

Details

The rule that makes this safe against run order: each writer's patch must contain only the keys it owns. `data_check` patches everything except code; `code_check` patches only code. Overlapping keys would still clobber.

Value

(invisibly) the manifest path

match_reported_output *Match reported statistical tests against the extracted analysis output*

Description

Recomposes the statistics a paper REPORTS (from `paper$eq`, grouped by the `grp_id` that ties one reported test's numbers together) into whole tests, then checks whether each test's component values co-occur in a single analysis of the paper's extracted OUTPUT (reproducibility_check's statistical output). A test is matched on its whole signature, not lone numbers, so a match means the reported result actually appears in the reproducible output rather than a value coinciding by chance. The result carries, per matched test, which output file and analysis produced it (provenance).

Usage

```
match_reported_output(
  paper,
  output,
  include_tables = FALSE,
  min_components = 1L
)
```

Arguments

paper	a paper object (its <code>\$eq</code> is recomposed), or an eq data frame
output	the extracted output: a <code>stat_results_long()</code> data frame, or the <code>stat_output</code> list a reproducibility_check result carries (its per-file <code>\$long</code> tables are combined, keeping each file's provenance)

- `include_tables` if TRUE and `paper` is a paper object, also build tests from statistics reported only in a results table's cells (e.g. from Grobid's table parsing – see `.tei_table_contents()` in `R/import-grobid.R` and the 5-tier header/caption/value-shape/row/matrix approach in `R/match-table.R`'s `.table_tests()`) and add them to the set checked against output. Default FALSE: table-derived tests are opt-in, currently used only by `reproducibility_check`, so every other caller's matching behaviour is unchanged.
- `min_components` a test must have at least this many recognised components to be assessed (default 1, so a lone reported statistic – a bare correlation $r = -.27$ or $d = 0.40$ with no accompanying p/CI – is still checked against the output, TYPE-GATED the same as any other component: it only matches an output cell of the SAME family, at the site with the most co-occurring components, via `val_in()`). This carries a higher coincidence risk than a multi-component test (a single r rounding to the reported value at some site is more likely than a whole signature doing so by chance), so a single-component match is worth reading with that in mind – set to 2 to exclude single-component rows entirely, as this function used to by default.

Value

a data.frame, one row per recomposed reported test, with: `text_id`, `grp_id`, `reported` (the recomposed test as text, e.g. "W=183.5 p=.791 rb=-0.16"), `n_components`, `n_matched` (components found in the best-matching output analysis), `found` (logical), `match_values` (the matched components as "name=value" pairs), `not_matched` (the UNMATCHED components of that same test, same "name=value" form – always populated alongside `match_values` for a partial match, so exactly which claim failed to reproduce is visible, not just a count), `source_file / analysis` (provenance of the match), `confidence` ("full" all components matched / "partial" / "none"), and `plausible_split` – NA unless this test's grouping came from `.regroup_by_evidence()` re-deriving test boundaries from OUTPUT co-occurrence rather than trusting `extract_tests()`'s text-only guess (see that function's own header comment): TRUE when a component split away from its original reported neighbours landed at a site sharing a `row_label` token with theirs (the same underlying variable, reached via a different analysis – plausible), FALSE when no such link was found (the split is NOT hidden or downgraded, only flagged, so a reader can judge the match's plausibility from the `source_file/analysis` of every piece involved). Attribute "summary" holds the roll-up.

mendeley_info

Retrieve info from Mendeley Data by URL

Description

Retrieve info from Mendeley Data by URL

Usage

```
mendeley_info(mendeley_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

mendeley_url	a Mendeley Data URL or DOI, or a table containing them (e.g., as created by mendeley_links())
id_col	the index or name of the column that contains Mendeley Data URLs, if mendeley_url is a table
pb	a progress bar passed from another function
cache	if TRUE, reuse a previously cached listing for a dataset already looked up (see repo_info_cache()) instead of re-querying the API. Off by default.

Value

a data frame of information

Examples

```
## Not run:
mendeley_info("https://doi.org/10.17632/vjtxybrc28.1")

## End(Not run)
```

mendeley_links	<i>Find Mendeley Data Links in Papers</i>
----------------	---

Description

Get all Mendeley Data links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE mention (a DOI like "10.17632/vjtxybrc28.1" is routinely cited without any URL scheme at all) that the source PDF/HTML never encoded as an actual hyperlink – the url table only ever contains links the source document itself made clickable. Same two-tier approach [zenodo_links\(\)](#) uses.

Usage

```
mendeley_links(paper)
```

Arguments

paper	a paper object or paperlist object
-------	------------------------------------

Value

a table with the Mendeley Data url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
mendeley_links(psychsci)

## End(Not run)
```

metacheck_cache_info *Show where metacheck's caches live and how big they are*

Description

Prints the location and size of all three on-disk caches — the downloaded repository-FILE cache, the repository-LISTING cache (dataset/record metadata, off by default – see [repo_info_cache\(\)](#)), and the LLM-response cache — so they are never hidden. By default each sits in a folder in the current working directory (.metacheck_repo_cache, .metacheck_repo_info_cache, and .metacheck_llm_cache); relocate all three with options(metacheck.cache.dir = "/some/path").

Usage

```
metacheck_cache_info()
```

Value

a data.frame (cache, path, size_mb), invisibly.

See Also

[repo_cache_dir\(\)](#), [repo_cache_clear\(\)](#), [repo_info_cache\(\)](#), [repo_info_cache_clear\(\)](#), [llm_cache_clear\(\)](#)

Examples

```
## Not run:
metacheck_cache_info()

## End(Not run)
```

module_help	<i>Get Module Help</i>
-------------	------------------------

Description

See the help files for a module by name (get a list of names from module_list())

Usage

module_help(module = NULL)

Arguments

module the name of a module or path to a module

Value

the help text

Examples

module_help("marginal")

module_info	<i>Get module information</i>
-------------	-------------------------------

Description

Get module information

Usage

module_info(module)

Arguments

module the name of a module or path to a module

Value

a list of module info

Examples

module_info("all_p_values")

module_install	<i>Install modules from GitHub</i>
----------------	------------------------------------

Description

Downloads a public GitHub repository that holds one or more metacheck modules and saves it in the metacheck user data directory, so its modules can be used by name in `module_run()`, `report()`, `module_list()` and `module_help()`, like the built-in modules.

Usage

```
module_install(repo, ref = NULL, overwrite = FALSE)
```

Arguments

repo	GitHub repository in "owner/repo" format, or its URL
ref	the branch, tag or commit to install (defaults to the repository's default branch)
overwrite	whether to replace the repository if it is already installed

Details

A module is an .R file with a roxygen header, as made by `module_template()`. Every such file in the top folder of the repository, or in a folder called `modules/`, is installed as a module. Other files (helper code, data, tests) are saved with the repository, but are not listed as modules.

A module can run other modules with `module_run()` and include their results in its own report with `module_report()`, so running one module (e.g. `report(paper, modules = "my_check")`) gives one report for all of its checks.

A module can state the lowest metacheck version it needs with a line `#' @metacheck 0.3.4` in its header; it then refuses to install or run with an older metacheck. External modules should only use exported metacheck functions: internal functions can change without notice.

Nothing is saved when a module in the repository has the same name as a built-in module, a module from another installed repository, or another module in the same repository.

Modules are saved in the metacheck user data folder, which belongs to the user who installs them. To share installed modules between users (for example with the user a Shiny Server runs apps as), set the environment variable `METACHECK_MODULES_DIR` to a folder all of them can read, both when installing and when using the modules.

Installing a module means running its code whenever you use it: only install modules from repositories you trust.

Value

a data frame of the installed repository (see `module_installed()`), invisibly

See Also

`module_update()`, `module_remove()`, `module_installed()`

Examples

```
## Not run:
module_install("owner/repo")
module_run(demopaper(), "module_name")

## End(Not run)
```

module_installed	<i>List installed module repositories</i>
------------------	---

Description

Lists the GitHub repositories installed with `module_install()`. Use `module_list()` to see their modules together with the built-in ones.

Usage

```
module_installed()
```

Value

a data frame with one row per repository: repo, the ref (branch, tag or commit) it was installed from, the sha of the installed commit, when it was installed, its modules, and the local path

Examples

```
module_installed()
```

module_list	<i>List modules</i>
-------------	---------------------

Description

List modules

Usage

```
module_list(
  module_dir = system.file("modules", package = "metacheck"),
  external = TRUE
)
```

Arguments

module_dir	the directory to search for modules (defaults to the built-in modules)
external	whether to also list the modules installed from GitHub with <code>module_install()</code>

Value

a data frame of modules; source is the GitHub repository of an installed module, or NA

Examples

```
mods <- module_list()
```

module_remove	<i>Remove installed modules</i>
---------------	---------------------------------

Description

Deletes a repository of modules installed with `module_install()`.

Usage

```
module_remove(repo)
```

Arguments

repo GitHub repository in "owner/repo" format

Value

TRUE invisibly if removed, FALSE if not installed

Examples

```
## Not run:  
module_remove("owner/repo")  
  
## End(Not run)
```

module_report	<i>Report from module output</i>
---------------	----------------------------------

Description

Report from module output

Usage

```
module_report(module_output, header = 3)
```

Arguments

module_output the output of a module_run()
header header level (default 2)

Value

text

Examples

```
paper <- demopaper()  
op <- module_run(paper, "stat_p_exact")  
module_report(op) |> cat()
```

module_run	<i>Run a module</i>
------------	---------------------

Description

Run a module

Usage

```
module_run(paper, module, ...)
```

Arguments

paper a paper object or a list of paper objects, or NULL to check only local files (a module's local_path argument)
module the name of a module or path to a module to run on this object
... further arguments to the module (e.g., arguments for the llm() function like params); these will override any arguments in the module

Value

a list of the returned table and report text

Examples

```
## Not run:  
psychsci <- papers_load("psychsci", cache = TRUE)  
module_run(psychsci[[1]], "all_p_values")  
  
## End(Not run)
```

module_template	Create a Module from a Template
-----------------	---------------------------------

Description

Create a Module from a Template

Usage

```
module_template(module_name, path = "./modules")
```

Arguments

module_name	The short name of the module (should contain only letters, numbers, and _)
path	The path of the directory to save the module in (defaults to a directory called "modules" in the working directory)

Value

the file path (invisibly)

module_update	Update installed modules
---------------	--------------------------

Description

Downloads the latest commit of the branch or tag each repository was installed from (see [module_install\(\)](#)). A repository that is already up to date is not downloaded again.

Usage

```
module_update(repo = NULL)
```

Arguments

repo	GitHub repository in "owner/repo" format, or NULL to update every installed repository
------	--

Value

a data frame of the installed repositories (see [module_installed\(\)](#)), invisibly

Examples

```
## Not run:  
module_update()  
  
## End(Not run)
```

openalex_doi	<i>OpenAlex info from DOI</i>
--------------	-------------------------------

Description

See details for a list of root-level fields that can be selected.

Usage

```
openalex_doi(doi, select = NULL)
```

Arguments

doi	the DOI of the paper to get info for
select	a vector of fields to return, NULL returns all

Details

See <https://docs.openalex.org/api-entities/works/work-object> for explanations of the information you can retrieve about works.

Root-level fields for the select argument:

- id
- doi
- title
- display_name
- publication_year
- publication_date
- ids
- language
- primary_location
- type
- type_crossref
- indexed_in
- open_access
- authorships
- institution_assertions
- countries_distinct_count
- institutions_distinct_count
- corresponding_author_ids
- corresponding_institution_ids

- apc_list
- apc_paid
- fwc
- has_fulltext
- fulltext_origin
- cited_by_count
- citation_normalized_percentile
- cited_by_percentile_year
- biblio
- is_retracted
- is_paratext
- primary_topic
- topics
- keywords
- concepts
- mesh
- locations_count
- locations
- best_oa_location
- sustainable_development_goals
- grants
- datasets
- versions
- referenced_works_count
- referenced_works
- related_works
- abstract_inverted_index
- abstract_inverted_index_v3
- cited_by_api_url
- counts_by_year
- updated_date
- created_date

Value

list with DOIs and info

Examples

```
doi <- "10.7717/peerj.4375"
## Not run:
oa_info <- openalex_doi(doi)
oa_info <- openalex_doi(doi, "title")

## End(Not run)
```

openalex_query	<i>Look up a reference in OpenAlex</i>
----------------	--

Description

Look up a reference in OpenAlex

Usage

```
openalex_query(title, source = NA, authors = NA, strict = TRUE)
```

Arguments

title	The title of the work
source	The source (journal or book)
authors	The authors
strict	Whether to return NULL or the best match if there isn't a single match

Value

A data frame with citation info

Examples

```
## Not run:
openalex_query("Sample Size Justification", "Collabra Psychology")

## End(Not run)
```

osf_api_check	<i>Check OSF API Server Status</i>
---------------	------------------------------------

Description

Check the status of the OSF API server.

Usage

```
osf_api_check(  
  osf_api = getOption("metacheck.osf.api"),  
  on_error = c("stop", "warn", "ignore")  
)
```

Arguments

osf_api	the OSF API to use (e.g., "https://api.osf.io/v2")
on_error	whether to stop, warn, or ignore errors

Details

The OSF API server is down a lot, so it's often good to check it before you run a bunch of OSF functions. When the server is down, it can take several seconds to return an error, so scripts where you are checking many URLs can take a long time before you realise they aren't working.

You can only make 100 API requests per hour, unless you authorise your requests, when you can make 10K requests per day. The osf functions in metacheck often make several requests per URL to get all of the info. You can authorise them by creating an OSF token at <https://osf.io/settings/tokens> and including the following line in your .Renviron file:

```
OSF_PAT="replace-with-your-token-string"
```

Value

the OSF status

Examples

```
osf_api_check()
```

osf_app	<i>Launch OSF Download App</i>
---------	--------------------------------

Description

Launch the OSF app: enter an OSF user or project ID, search and tick the projects you want, then download them with `osf_file_download()`.

Usage

```
osf_app(quiet = FALSE, ...)
```

Arguments

quiet	whether to show debugging messages in the console
...	arguments to pass to shiny::runApp

Details

Listing a user's projects uses `osf_user_projects()`, and a single project ID is shown as a one-row list so both routes end in the same place. Private projects need a token, which is entered in the app and set for that session only with `osf_pat()`.

Value

NULL (invisibly)

Examples

```
## Not run:
osf_app()

## End(Not run)
```

osf_cache_clear	<i>Clear the session cache of OSF listings</i>
-----------------	--

Description

Clear the session cache of OSF listings

Usage

```
osf_cache_clear()
```

Value

the number of cached listings removed, invisibly

osf_check_id	<i>Check OSF IDs</i>
--------------	----------------------

Description

Check if strings are valid OSF IDs, URLs, or waterbutler IDs. Basically an improved wrapper for `osfr::as_id()` that returns NA for invalid IDs in a vector.

Usage

```
osf_check_id(osf_id)
```

Arguments

`osf_id` a vector of OSF IDs or URLs

Value

a vector of valid IDs, with NA in place of invalid IDs

Examples

```
osf_check_id("pngda")
osf_check_id("osf.io/pngda")
osf_check_id("https://osf.io/pngda")
osf_check_id("https://osf .io/png da") # rogue whitespace
osf_check_id("pnda") # invalid
```

osf_delay	<i>Set the OSF delay</i>
-----------	--------------------------

Description

Sometimes the OSF gets fussy if you make too many calls, so you can set a delay of a few seconds before each call. Use `osf_delay()` to get or set the OSF delay.

Usage

```
osf_delay(delay = NULL)
```

Arguments

`delay` the number of seconds to wait between OSF calls

Examples

```
osf_delay()
```

osf_file_download	<i>Download all OSF Project Files</i>
-------------------	---------------------------------------

Description

Creates a directory for the OSF ID and downloads all of the files using a folder structure from the OSF project nodes and file storage structure. Returns (invisibly) a data frame with file info.

Usage

```
osf_file_download(
  osf_id,
  download_to = ".",
  max_file_size = NULL,
  max_download_size = NULL,
  max_folder_length = Inf,
  ignore_folder_structure = FALSE,
  mode = c("all", "select", "files", "zip"),
  unzip = TRUE,
  metadata = TRUE,
  osf_pat = NULL,
  pb = NULL
)
```

Arguments

osf_id	an OSF ID or URL. A <i>user</i> ID (the GUID in a profile URL, e.g. "4i578" for https://osf.io/4i578) downloads every project that user contributes to, so a whole account can be archived in one call. To choose a subset instead, pass the table from <code>osf_user_projects()</code> (filtered however you like) – any data frame with an <code>osf_id</code> column works.
download_to	path to download to. Each project is saved in its own folder here, named after its OSF ID. Downloading the same project again reuses that folder and fetches only what is missing, so re-running after a partial download resumes it instead of making a second copy.
max_file_size	largest single file to download, in MB. NULL (the default) means no limit: this function exists to retrieve a repository in full, and someone archiving their own work wants their own files whatever size they are. Set a number to skip anything larger, which is useful when looking at a repository that is not yours.
max_download_size	largest total per project, in MB. NULL (the default) means no limit, for the same reason.

max_folder_length	maximum folder name length (set to make sure paths are <260 character on some Windows OS)
ignore_folder_structure	if TRUE, download all files into a single folder
mode	what you want from the repository: • "all" (the default) – the whole thing, as fast as the OSF allows. It walks the component tree and takes one archive per component, never listing individual files. Listing is what makes a large repository slow: for ManyLabs2 it ran over 15 minutes before fetching anything, while this route retrieved the whole 1.9 GB project in under 7 minutes. Files on a linked add-on (GitHub, Dropbox) are not in any OSF archive, so those are fetched individually. Size limits do not apply, because there is no file list to filter, and neither does resuming: running it again downloads every archive again. Knowing what to skip would mean reading each archive's index without fetching it, and the OSF does not allow that (a HEAD returns 501 and a Range request returns the whole file, so zip_peek() cannot read it). Use "select" if you need to resume. • "select" – list every file first, apply max_file_size and max_download_size, then fetch what survives, checking each file against the size the OSF reported. Use this when you want part of a repository, or want each file verified. • "zip" – like "select", but transports whole nodes as archives once the listing is done. Fewer requests than "select", though not faster. • "files" – an old name for "select", still accepted.
unzip	if TRUE and mode = "zip", unzip the downloaded archive into the output folder; if FALSE, keep the zip file as-is
metadata	whether to also retrieve the parts of the project that are not files, into an _osf_metadata folder inside the project's folder: wiki pages as Markdown (current version only), the activity log as logs.csv, the project's title, description, tags, licence and contributors as metadata.json, and a readable README.md summarising them. Costs about four extra API requests per project.
osf_pat	an OSF personal access token, needed to reach private projects and to raise the API rate limit. Defaults to whatever osf_pat() returns (the OSF_PAT environment variable unless set otherwise); passing it here sets it for the rest of the session. See osf_pat() for how to create one.
pb	a progress bar passed from another function

Details

Some differences may exist because the OSF allows longer file names with characters that may not be allowed on a file system, so these are cleaned up when downloading.

Everything is downloaded by default: no size limits apply, because this function exists to retrieve a repository in full and someone archiving their own work wants their own files whatever size they are. Limits are available for when that is not what you want – looking at somebody else's repository, say. In the default mode = "files", files over max_file_size are omitted individually,

while `max_download_size` is an all-or-nothing gate for the remaining repository total. Omitted files are reported as messages and appear in the returned data frame with `downloaded = FALSE` and `attempted = FALSE`.

In mode = "zip", OSF's Waterbutler API serves a folder as one generated zip archive, which is far fewer requests than fetching each file. That endpoint covers one node's `osfstorage` at a time, so a project whose files sit in components, or on a linked add-on such as GitHub or Dropbox, is not one archive: one archive is requested per node that holds `osfstorage` files, and anything no archive can hold is downloaded individually, so the whole project is still retrieved. In this mode, `max_download_size` is a budget for the whole project. The OSF does not report an archive's size in advance, so a node's archive is taken only when its listed files fit in what is left of the budget, and its download is stopped if it grows past that. The files of a node whose archive does not fit, or was stopped, are downloaded individually, smallest first, as many as fit in the budget. `max_file_size` cannot filter files inside an archive before download. Archives can either be kept as zips or unzipped after download.

Value

data frame of file info, one row per file (`osf_id` is each file's own ID). It carries `download_path`, the absolute folder the project was saved in, plus `osf_project` and `osf_url` identifying the project, so the result can be passed straight to `zenodo_upload()`. `downloaded`, `size_on_disk`, and `attempted` report the verification described above.

In mode = "all" there is no file listing, so the table has one row per **node** instead: `folder`, `osf_project`, `osf_url`, `title`, `files` (how many arrived), `bytes`, `download_path`, and `downloaded`. It can still be passed to `zenodo_upload()`, which uses `download_path`.

Downloading a whole account, or part of one

Given a user ID, every project that user contributes to is downloaded. Components are not listed separately, because downloading a project already brings its components' files with it, at any depth.

To download only some of them, list the projects first with `osf_user_projects()`, filter that table however you like, and pass it back:

```
projects <- osf_user_projects("4i578")
subset(projects, public) |> osf_file_download(download_to = "my_osf")
```

Checking the download

Every file that was meant to arrive is checked against the file system afterwards, and against the size the OSF reported for it, so a transfer that failed silently or wrote a truncated file is reported rather than passing as complete. Two columns record the outcome: `downloaded` is TRUE only for a file actually present at the right size, and `attempted` is FALSE for a file deliberately excluded by `max_file_size` or `max_download_size`. Files that were attempted but did not arrive raise a warning naming them.

Examples

```
## Not run:
osf_file_download("6nt4v")
```

```
# download everything one user has, then archive it on Zenodo
osf_file_download("4i578", download_to = "my_osf",
                  max_file_size = NULL) |>
  zenodo_upload()

## End(Not run)
```

osf_get_all_pages	<i>Get All OSF API Query Pages</i>
-------------------	------------------------------------

Description

OSF API queries only return up to 10 items per page, so this helper functions checks for extra pages and returns all of them

Usage

```
osf_get_all_pages(url, page_end = Inf)
```

Arguments

url	the OSF API URL
page_end	The last page to get

Value

a table of the returned data. When the request could not be completed (e.g. a private, embargoed, withdrawn, or deleted resource, or a network failure), this is an empty `list()` with an `osf_error` attribute set to "forbidden", "not_found", "gone", or "request_failed" (a bare `NULL` cannot carry attributes in R, so it would silently discard this) – callers that only check `length(result) == 0` keep working unchanged, and callers that need to tell "inaccessible" apart from "genuinely empty" can check `attr(result, "osf_error")`.

Examples

```
# get the 20 newest preprints
## Not run:
osf_api <- getOption("metacheck.osf.api")
url <- sprintf("%s/preprints/?search=date_created-desc", osf_api)
preprints <- osf_get_all_pages(url, 2)

## End(Not run)
```

osf_info

*Retrieve info from the OSF by ID***Description**

Repository listings are cached for the session (keyed by URL and recursive), so repeated calls do not re-query the OSF API; clear the cache with `osf_cache_clear()` or disable it with `options(metacheck.osf.cache = FALSE)`.

Usage

```
osf_info(osf_url, id_col = 1, recursive = FALSE, pb = NULL, cache = FALSE)
```

Arguments

<code>osf_url</code>	an OSF ID or URL, or a table containing them
<code>id_col</code>	the index or name of the column that contains OSF IDs or URLs, if <code>id</code> is a table
<code>recursive</code>	whether to retrieve all children
<code>pb</code>	a progress bar passed from another function
<code>cache</code>	if TRUE, also reuse a PERSISTENT, on-disk cached listing per id (see <code>repo_info_cache()</code>) that survives a restart of the script – separate from, and in addition to, this function’s own in-memory, session-only cache described above. Off by default.

Value

a data frame of information

Examples

```
## Not run:
# get info on one OSF node
osf_info("pngda")

# also get child nodes and files
osf_info("https://osf.io/6nt4v", recursive = TRUE)

## End(Not run)
```

osf_links

*Find OSF Links in Papers***Description**

Get all OSF links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE mention like "osf.io/gms8z/" that the source PDF/HTML never encoded as an actual hyperlink (common in PDF-converted papers, where a plain-text URL mention loses its link formatting) — the url table only ever contains links the source document itself made clickable, so a repository this important cannot rely on that alone. Same two-tier approach `github_links()` already uses for GitHub.

Usage

```
osf_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the OSF url in the first (href) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
osf_links(psychsci)

## End(Not run)
```

osf_pat

*Set or get the OSF personal access token***Description**

Use `osf_pat()` to get the token used to authorise OSF API requests, or `osf_pat("your-token")` to set it for the rest of the session.

Usage

```
osf_pat(pat = NULL)
```

Arguments

pat the token to set, or NULL to get the current token

Details

Without a token you can make 100 API requests per hour; with one you can make 10,000 per day, so a token is worth setting before downloading whole projects or listing everything a user has.

To create one, sign in at <https://osf.io/settings/tokens>, click "Create token", give it a name, and tick the `osf.full_read` scope (add `osf.full_write` only if you also intend to write to the OSF). Copy the token immediately, as the OSF does not show it again.

The usual way to store it is a line in your `.Renviron` file (open it with `usethis::edit_r_environ()`), which is read every time R starts:

```
OSF_PAT="replace-with-your-token-string"
```

Setting the token with `osf_pat()` overrides that for the current session only, which is useful when you cannot edit `.Renviron` or want to use a different account for one run.

Value

the current token (character; "" when none is set)

Examples

```
osf_pat() # returns "" unless a token is set
```

<code>osf_preprint_list</code>	<i>Get A list of preprints from the OSF</i>
--------------------------------	---

Description

Get A list of preprints from the OSF

Usage

```
osf_preprint_list(
  provider = NULL,
  date_created = NULL,
  date_modified = NULL,
  page_start = 1,
  page_end = page_start
)
```

Arguments

<code>provider</code>	a vector of the preprint providers, e.g. <code>psyarxiv</code> , <code>socarxiv</code> , <code>edaxriv</code> (see https://osf.io/preprints/discover)
<code>date_created</code>	a single date or a vector of two date (min and max)
<code>date_modified</code>	a single date or a vector of two date (min and max)
<code>page_start</code>	the first page of 10 entries
<code>page_end</code>	the last page of 10 entires to read

Value

a table of preprint info

Examples

```
## Not run:
dc <- c("2025-09-01", "2025-10-01")
pp <- osf_preprint_list("psyarxiv", date_created = dc)
files <- pp$primary_file

## End(Not run)
```

osf_type

Get OSF GUID Type

Description

Get OSF GUID Type

Usage

```
osf_type(guid)
```

Arguments

guid the 5-letter GUID

Value

the type, or "inaccessible" when the GUID is a validly-formed OSF ID but the resource itself could not be reached (private, embargoed, withdrawn, or deleted) – distinct from NA, which means the input was not a valid OSF ID at all

Examples

```
# osf_type("pngda")
```

osf_user_projects	<i>List the Projects Belonging to an OSF User</i>
-------------------	---

Description

Every project an OSF user contributes to, as a table you can read and filter before downloading anything. Pass the whole table, or any subset of it, to `osf_file_download()`.

Usage

```
osf_user_projects(user_id, pb = NULL)
```

Arguments

user_id	an OSF user ID (the 5-character GUID in a profile URL, e.g. "4i578" for https://osf.io/4i578)
pb	a progress bar passed from another function

Details

`/users/{id}/nodes/` returns each node the user can see, which includes components as separate rows alongside the projects that contain them. Downloading a project pulls in its components' files automatically (the folder structure in `osf_file_download()` nests them, at any depth), so listing a component separately would only download the same files twice. Each node is therefore reduced to its root (relationships\$root, which `.osf_node_data()` exposes as the project column); a node with no root recorded stands as its own project.

No file counts or sizes are reported. They would cost an extra request per project and could only ever count files at the project's own root, missing everything inside folders and components – while the download itself already lists each project in full, so the true size is discovered there anyway.

Value

a data frame with one row per project: `osf_id`, `name`, `category`, `public`, and `osf_url`. Zero rows when the user has no projects that could be listed.

Examples

```
## Not run:
# see what there is
projects <- osf_user_projects("4i578")

# download only the public ones
projects |>
  subset(public) |>
  osf_file_download(download_to = "my_osf")

# or pick by name
projects |>
```

```
subset(grepl("ManyLabs", name)) |>
  osf_file_download(download_to = "manylabs")

## End(Not run)
```

paper_id

Get Paper IDs

Description

Get Paper IDs

Usage

```
paper_id(paper)
```

Arguments

paper a paper or paperlist

Value

a vector of paper_ids

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
paper_id(psychsci)

## End(Not run)
```

paper_table

Paper tables

Description

Return a table from a paper object or concatenate tables across a list of paper objects.

Usage

```
paper_table(paper, table, cols = NULL)
```

Arguments

paper a paper or paperlist
 table a table name
 cols the columns to return from the table (default all columns)

Value

a merged table

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
biblio <- paper_table(psychsci[1:10], "bib")
xrefs <- paper_table(psychsci[1:10], "xref")

## End(Not run)
```

paper_validate	<i>Validate a Paper Object</i>
----------------	--------------------------------

Description

Checks if a paper object conforms to the JSON schema.

Usage

```
paper_validate(paper)
```

Arguments

paper	a paper object
-------	----------------

Value

TRUE or error

Examples

```
paper <- list(paper_id = "Not a paper object")
tryCatch(
  paper_validate(paper),
  error = \(e) print(e$message)
)

paper <- demopaper()
paper_validate(paper)
```

paper_write	<i>Write paper</i>
-------------	--------------------

Description

Save a paper as a JSON file.

Usage

```
paper_write(paper, file_name = NULL, save_path = ".", schema_version = NULL)
```

Arguments

paper	a paper object
file_name	the name of the file (if NULL, defaults to the paper_id)
save_path	the directory to save the JSON file in
schema_version	NULL (the default) saves the paper object as is; "12.0" saves a bibr export schema 12.0 file, for a paper read from a bibr 12.0 export or converted with <code>grobid_to_bibr(schema_version = "12.0")</code> . The file keeps the paper's extraction block (a bibr export keeps bibr as its producer and the time bibr extracted it) and names metacheck as the converter.

Value

the path to the JSON file

Examples

```
## Not run:
paper <- demopaper()
paper$info$title <- "New title"
paper_write(paper, "new_paper")

## End(Not run)
```

papers_available	<i>List available paper corpora</i>
------------------	-------------------------------------

Description

Queries the GitHub Releases API for the scienceverse/papers repository and returns a table of available paper corpora that can be loaded with [papers_load\(\)](#).

Usage

```
papers_available(repo = "scienceverse/papers")
```

Arguments

repo GitHub repository in "owner/repo" format

Value

a data frame with columns name, tag, size_mb, and cached

Examples

```
## Not run:
papers_available()

## End(Not run)
```

papers_load	<i>Load a paper corpus</i>
-------------	----------------------------

Description

Downloads a paper corpus RDS file from the scienceverse/papers GitHub repository and loads it into R as a paperlist object. By default the file is downloaded to a temporary location and discarded after loading, which is the right choice for one-off use. Set cache = TRUE to save it permanently in the user data directory instead, so subsequent calls reuse the cached copy rather than re-downloading. Use [papers_remove\(\)](#) to delete a cached corpus.

Usage

```
papers_load(
  name,
  repo = "scienceverse/papers",
  cache = FALSE,
  overwrite = FALSE
)
```

Arguments

name name of the corpus, without the .rds extension (see [papers_available\(\)](#))

repo GitHub repository in "owner/repo" format

cache whether to save the corpus to the user data directory for reuse across sessions, instead of downloading it fresh each time

overwrite whether to re-download if already cached (only relevant when cache = TRUE)

Value

a paperlist object

Examples

```
## Not run:
collabra <- papers_load("collabra")

## End(Not run)
```

papers_metadata	<i>Get a paper corpus's Dublin Core metadata</i>
-----------------	--

Description

Fetches and parses `metadata.json` for a corpus from the `scienceverse/papers` GitHub repository. This file documents provenance that is not stored in the corpus `.rds` itself: how the corpus was built, what years/license it covers, and so on. See each corpus's `README.md` on the [papers repository](#) for a fuller narrative description, including known gaps and data-quality caveats.

Usage

```
papers_metadata(name, repo = "scienceverse/papers")
```

Arguments

name	name of the corpus (see papers_available())
repo	GitHub repository in "owner/repo" format

Value

a named list of the corpus's Dublin Core metadata fields (colons in the original `dc:field` names are replaced with underscores, e.g. `dc:title` becomes `dc_title`)

Examples

```
## Not run:
papers_metadata("jdm")$dc_coverage

## End(Not run)
```

<code>papers_remove</code>	<i>Remove a cached paper corpus</i>
----------------------------	-------------------------------------

Description

Deletes the locally cached RDS file for a corpus to free disk space.

Usage

```
papers_remove(name)
```

Arguments

<code>name</code>	name of the corpus (see papers_available())
-------------------	--

Value

TRUE invisibly if deleted, FALSE if not cached

Examples

```
## Not run:  
papers_remove("collabra")  
  
## End(Not run)
```

<code>path_sanitize</code>	<i>Sanitize File Path</i>
----------------------------	---------------------------

Description

Make sure user-input file names are not problematic.

Usage

```
path_sanitize(  
  path,  
  replacement = "_",  
  remove_whitespace = TRUE,  
  keep_sep = TRUE  
)
```

Arguments

path	the path to sanitize (can be a vector of paths)
replacement	the character to replace invalid characters with
remove_whitespace	whether to include whitespace as a problem
keep_sep	whether to keep the path separator /

Value

the sanitized vector

Examples

```
path <- "/My Files/x<y.pdf"
path_sanitize(path)
path_sanitize(path, replacement = "~")
path_sanitize(path, remove_whitespace = FALSE)
path_sanitize(path, keep_sep = FALSE)
```

plural	<i>Pluralise</i>
--------	------------------

Description

Helper function for conditional plurals. For example, if you want to return "1 error" or "2 errors", you can use this in a sprintf().

Usage

```
plural(n, singular = "", plural = "s")
```

Arguments

n	the number
singular	the word or ending when n = 1
plural	the word or ending n != 1

Value

a string

Examples

```
n <- 0:3
sprintf("I have %d friend%s", n, plural(n))
sprintf("I have %d %s", n, plural(n, "octopus", "octopi"))
```

psycharchives_file_download

Retrieve public file list from PsychArchives by URL

Description

Lists the publicly retrievable bitstreams of one or more PsychArchives items via the DSpace REST API, without downloading them. Each row carries an absolute `file_url` (the bitstream retrieve endpoint) so the actual bytes are fetched later by `download_repo_files()`, the same deferred path used for Zenodo and OSF. Restricted files are omitted by the API and never appear here.

Usage

```
psycharchives_file_download(pa_url, pb = NULL, cache = FALSE)
```

Arguments

<code>pa_url</code>	a vector of PsychArchives URLs
<code>pb</code>	a progress bar passed from another function
<code>cache</code>	if TRUE, reuse a cached listing from a prior call for the same item instead of re-querying PsychArchives (see <code>repo_info_cache()</code>); default FALSE

Value

a data frame of file information (one row per public bitstream)

psycharchives_info

Retrieve info from PsychArchives by URL

Description

Retrieve info from PsychArchives by URL

Usage

```
psycharchives_info(pa_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

<code>pa_url</code>	a PsychArchives URL (item page, handle, or hdl.handle.net link), or a table containing them (e.g., as created by <code>psycharchives_links()</code>)
<code>id_col</code>	the index or name of the column that contains PsychArchives URLs, if <code>pa_url</code> is a table
<code>pb</code>	a progress bar passed from another function
<code>cache</code>	if TRUE, reuse a cached listing from a prior call for the same item instead of re-querying PsychArchives (see <code>repo_info_cache()</code>); default FALSE

Value

a data frame of information

Examples

```
## Not run:
psycharchives_info("https://hdl.handle.net/20.500.12034/17526")

## End(Not run)
```

psycharchives_links	<i>Find PsychArchives Links in Papers</i>
---------------------	---

Description

Get all PsychArchives links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE mention (a handle like "20.500.12034/17526" is routinely cited without any URL scheme at all) that the source PDF/HTML never encoded as an actual hyperlink — the url table only ever contains links the source document itself made clickable. Same two-tier approach `github_links()` uses for GitHub.

Usage

```
psycharchives_links(paper)
```

Arguments

paper	a paper object or paperlist object
-------	------------------------------------

Value

a table with the PsychArchives url in the first (href) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
psycharchives_links(psychsci)

## End(Not run)
```

pubpeer_comments	<i>Get Pubpeer Comments</i>
------------------	-----------------------------

Description

Takes a DOI, and retrieves information from pubpeer related to post-publication peer review comments.

Usage

```
pubpeer_comments(doi)
```

Arguments

doi	a vector of paper DOIs
-----	------------------------

Value

a dataframe with information from pubpeer

Examples

```
doi <- c(
  "10.1038/s41598-025-24662-9",
  "10.1177/0146167211398138"
)
pubpeer_comments(doi)
```

rbox_file_download	<i>Retrieve files from ResearchBox by URL</i>
--------------------	---

Description

Retrieve files from ResearchBox by URL

Usage

```
rbox_file_download(rb_url, pb = NULL)
```

Arguments

rb_url	a vector of ResearchBox URLs
pb	a progress bar passed from another function

Value

a data frame of information

rbox_info	<i>Retrieve info from ResearchBox by URL</i>
-----------	--

Description

Retrieve info from ResearchBox by URL

Usage

```
rbox_info(rb_url, id_col = 1, pb = NULL)
```

Arguments

rb_url	an ResearchBox URL, or a table containing them (e.g., as created by rbox_links())
id_col	the index or name of the column that contains ResearchBox URLs, if id is a table
pb	a progress bar passed from another function

Value

a data frame of information

Examples

```
## Not run:
# get info on one OSF node
rbox_info("https://researchbox.org/801")

## End(Not run)
```

rbox_links	<i>Find ResearchBox Links in Papers</i>
------------	---

Description

Get all ResearchBox links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE mention like "researchbox.org/ 801" that the source PDF/HTML never encoded as an actual hyperlink (common in PDF-converted papers, where a plain-text URL mention loses its link formatting) — the url table only ever contains links the source document itself made clickable. Same two-tier approach github_links() uses for GitHub.

Usage

```
rbox_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the ResearchBox url in the first (href) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
rbox_links(psychsci)

## End(Not run)
```

read	<i>Read in grobid XML or bibr JSON</i>
------	--

Description

Read in grobid XML or bibr JSON

Usage

```
read(file_path, include_images = FALSE, recursive = FALSE)
```

Arguments

file_path path to a single directory containing XML and/or JSON files, or a vector of XML/JSON paths

include_images whether to include images in the figures table of the paper object (they make object size larger, only relevant to bibr imports)

recursive whether to read files in subfolders (files should have unique paper_ids, or errors can occur)

Value

a paper or paperlist

read_r_output	<i>Extract statistical results from captured R console output</i>
---------------	---

Description

Parses the text an R script prints (as captured from `source(script, echo = TRUE)`) into tidy result tables, one per detected result block. Handles both one-line test output (`t.test`, `cor.test`, ...) and fixed-width text tables (`summary(lm)`, `aov`, `anova`). The result matches the shape of `read_stat_tables()`, so it flows into the same STATO typing and ISA-JSON export.

Usage

```
read_r_output(text, source_label = NA_character_, code_lines = NULL)
```

Arguments

<code>text</code>	the captured console output: a character vector of lines, or a single string with embedded newlines
<code>source_label</code>	optional label (e.g. the script name) recorded as the analysis for one-line results
<code>code_lines</code>	optional character vector of the script's source lines (one element per line), used to recover the source line of each result via the echoed statement text. <code>NULL</code> (default) skips line attribution.

Details

When `code_lines` is supplied (the script's own text, one element per source line – what a run's `code_text_list[[file]]` already is), each result is additionally tagged with the 1-based source line it came from, recovered by matching the echoed `> /+` statement text against the script. A single statement that prints several results (e.g. a for loop calling `t.test()` each iteration) yields several results sharing one line; these are distinguished by `line_seq` (1, 2, 3, ... within that line). Without `code_lines` the echo is not parsed and `line`/`line_seq` are `NA`.

Value

a list with one element per detected result block, each a list of analysis (the test/section label), title, data (a tidy data.frame), line (1-based source line, or `NA` when `code_lines` was not supplied), `line_seq` (1-based counter of results sharing that line, or `NA`), and `call_fn` (the recognised statistical function that produced the block, e.g. `"shapiro.test"`, or `""`) – the same structure `read_stat_tables()` returns, plus these three fields. `call_fn` lets `stato_type_column()` resolve statistic letters that are ambiguous in the printed output alone.

read_stat_tables	<i>Read the statistical result tables from a JASP, jamovi or notebook file</i>
------------------	--

Description

Opens a .jasp, .omv, .spv or .ipynb file and returns every result table as a tidy data frame together with the analysis it belongs to. No statistics knowledge is applied here — this is the lossless extraction layer; semantic typing happens downstream in [stat_output_json\(\)](#) / [stat_results_long\(\)](#).

Usage

```
read_stat_tables(path)
```

Arguments

path	path to a .jasp, .omv, .spv or .ipynb file
------	--

Details

Four sources are tried, in order of fidelity:

1. .jasp — the archive's own analyses.json, which holds the structured results;
2. .omv — jamovi's protobuf-serialised AnalysisResponse blobs, decoded natively (see `inst/schema/jamovi/PROVENANCE`);
3. .ipynb — the outputs saved inside each code cell of a Jupyter notebook (dispatched on extension; it is JSON, not a zip archive);
4. any archive — the rendered `index.html`, as a fallback when the above are absent or unreadable.

The structured sources are preferred because they carry each column's machine name even when its displayed header is blank (the HTML then yields an unnamed column), store values at full precision (a p of 0.000632 rather than the rendered "< .001"), and mark missing cells explicitly instead of as a rendered dash.

Value

a list with one element per result table, each a list of `analysis` (the nearest heading above the table), `title` (the table's own caption / first heading, when distinguishable), `data` (a `data.frame` of the table as rendered, header row as column names), and `table_index` (1-based ordinal position of this table among all tables in the rendered document — there is no source line to attach, since a JASP/jamovi analysis is GUI-produced). Empty list when the archive has no `index.html` or no tables.

ref_table	<i>Reference and DOI table</i>
-----------	--------------------------------

Description

Return a table with fixed DOIs and reference text from a paper object or concatenate tables across a list of paper objects.

Usage

```
ref_table(paper)
```

Arguments

paper a paper or paperlist

Value

a merged table

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
biblio <- ref_table(psychsci[[1]])

## End(Not run)
```

regcheck_compare	<i>Compare a Preregistration with a Paper via RegCheck</i>
------------------	--

Description

Sends the full text of a paper and the text of its (pre)registration to a **RegCheck** server, which uses an LLM to compare them dimension by dimension (e.g., sample size, hypotheses, exclusion criteria) and judge for each dimension whether the paper deviates from the registration.

Usage

```
regcheck_compare(
  paper_text,
  prereg_text = NULL,
  registration_id = NULL,
  client = c("ollama", "groq", "openai", "deepseek"),
  base_url = NULL,
  api_token = NULL,
```

```

    dimensions = NULL,
    reasoning_effort = "medium",
    poll_interval = 5,
    timeout = 3600
  )

```

Arguments

<code>paper_text</code>	the full text of the paper (single string)
<code>prereg_text</code>	the text of the preregistration (single string); provide exactly one of <code>prereg_text</code> or <code>registration_id</code>
<code>registration_id</code>	a clinical trial registration ID (NCT number) to compare against, as an alternative to <code>prereg_text</code>
<code>client</code>	the LLM client the RegCheck server should use: "ollama" (local), "groq", "openai", or "deepseek"
<code>base_url</code>	the base URL of the RegCheck server; defaults to <code>http://localhost:8000</code> for ollama and the hosted RegCheck app otherwise (can also be set with the <code>REGCHECK_BASE_URL</code> environment variable)
<code>api_token</code>	the RegCheck API token
<code>dimensions</code>	optional data frame with columns <code>dimension</code> and <code>definition</code> to override the server's default comparison dimensions
<code>reasoning_effort</code>	reasoning effort for OpenAI models ("low", "medium", "high"); ignored by other clients
<code>poll_interval</code>	seconds between polls of the task status
<code>timeout</code>	maximum seconds to wait for the comparison to complete

Details

By default this uses the locally running RegCheck server bundled in `metacheck` with Ollama (`client = "ollama"`, `http://localhost:8000`), so no text leaves your machine. Set `client` to `"groq"`, `"openai"` or `"deepseek"` to use the hosted RegCheck app instead; note that this sends the full paper and registration text to that server and the corresponding LLM provider.

All RegCheck servers (including a local one) require an API token. Set it in your `.Renviron` (use `usethis::edit_r_environ()`) as:

```
REGCHECK_API_TOKEN="your_token"
```

For a local server this must match the `REGCHECK_API_TOKEN` you configured when starting the server.

A comparison makes one LLM call per dimension and can take several minutes. The function polls the server until the comparison completes.

Value

a data frame with one row per compared dimension, with columns `dimension`, `deviation_judgement` ("yes", "no", or "missing"), `paper_summary`, `prereg_summary`, `deviation_information`, `paper_quotes`, and `prereg_quotes`. The raw server result is attached as the `"regcheck_result"` attribute.

Examples

```
## Not run:
paper <- demopaper()
paper_text <- paste(paper$text$text, collapse = " ")
prereg_text <- "We will collect 100 participants..."
rc <- regcheck_compare(paper_text, prereg_text)

## End(Not run)
```

regcheck_setup_local *Set up the local RegCheck server (manual Python path)*

Description

Creates a Python virtual environment inside the bundled RegCheck app directory, installs all dependencies, and downloads the NLTK data. Only needed if you are using the manual Python path – if you have Docker, use [regcheck_start_local\(\)](#) directly with method = "docker".

Usage

```
regcheck_setup_local(python = NULL)
```

Arguments

python	path to the Python 3.12+ executable; if NULL (default), tries python3 then python on the system PATH.
--------	---

Details**System requirements:**

- Python 3.12+ – <https://www.python.org/downloads/>
- Ollama – <https://ollama.com/download>, then run `ollama pull llama3.2` and `ollama pull nomic-embed-text-v2-moe` in a terminal.

Value

invisibly, the path to the virtual environment

regcheck_start_local	<i>Start the local RegCheck server</i>
----------------------	--

Description

Starts the bundled RegCheck server as a background process, either via Docker (recommended) or a manual Python virtual environment. The server runs at `http://localhost:8000` and is used automatically by `module_run(paper, "reg_check")` (the default client = "ollama").

Usage

```
regcheck_start_local(method = c("docker", "python"), model = NULL, port = 8000)
```

Arguments

method	"docker" (default) or "python"
model	Ollama language model to use. Defaults to NULL, which auto-selects the largest language model currently pulled in Ollama. Any model you have pulled with <code>ollama pull</code> can be used here – larger models generally produce better judgments.
port	port to run the server on (default 8000)

Details

Docker path (method = "docker", recommended): Requires **Docker** to be installed. The first run builds the image (~5 minutes); subsequent starts are instant. No Python installation needed.

Python path (method = "python"): Requires Python 3.12+ and Ollama. Run `regcheck_setup_local()` once first.

In both cases, Ollama must be running with the required models pulled. The embedding model is fixed; the language model can be any Ollama model you have pulled – use the largest one that fits your hardware:

```
ollama pull nomic-embed-text-v2-moe
ollama pull llama3.2          # or llama3.1, mistral, etc.
```

The API token is set automatically to "metacheck-local" – no manual token configuration needed.

Value

invisibly, the process object

regcheck_stop_local	<i>Stop the local RegCheck server</i>
---------------------	---------------------------------------

Description

Kills the background process started by [regcheck_start_local\(\)](#).

Usage

```
regcheck_stop_local()
```

Value

invisibly NULL

repo_cache_clear	<i>Delete the downloaded-repository file cache</i>
------------------	--

Description

Removes files fetched from data repositories by `data_check` / `code_check` from the on-disk cache (see [repo_cache_dir\(\)](#)). The cache only speeds up re-runs — anything deleted is simply re-downloaded when next needed — so clearing it is always safe; the usual reason is to reclaim disk space after a large corpus build.

Usage

```
repo_cache_clear(repo_url = NULL, quiet = FALSE)
```

Arguments

<code>repo_url</code>	optional character vector of repository URLs. When supplied, only those repositories' cached files are deleted; otherwise the entire cache is cleared.
<code>quiet</code>	if FALSE (default), report how much was freed.

Details

By default the whole cache is removed. Pass `repo_url` to remove only the cached files of specific repositories.

Value

the number of bytes freed (numeric), invisibly.

See Also

[repo_cache_dir\(\)](#), [repo_cache_size\(\)](#)

Examples

```
## Not run:
# clear everything
repo_cache_clear()

# clear one repository only
repo_cache_clear("https://osf.io/abcde")

## End(Not run)
```

repo_cache_dir

Locate the downloaded-repository file cache

Description

Files fetched from OSF/GitHub/Zenodo/ResearchBox by `data_check` and `code_check` are stored in a persistent on-disk cache so they are reused across modules and R sessions (never re-downloaded). This returns that cache's root directory.

Usage

```
repo_cache_dir()
```

Details

The cache is never cleared automatically — it has no size cap or expiry, so it is safe to reuse but grows without bound. Use [repo_cache_clear\(\)](#) to delete it (e.g. to reclaim disk space after a large corpus run).

The location defaults to `rappdirs::user_cache_dir("metacheck/repo_files")`; override it with `options(metacheck.repo_cache.dir = "/some/path")` (e.g. for a project-local cache).

Value

the cache root directory path (character), invisibly

See Also

[repo_cache_clear\(\)](#), [repo_cache_size\(\)](#)

Examples

```
## Not run:
repo_cache_dir()

## End(Not run)
```

repo_cache_size	<i>Size of the downloaded-repository file cache</i>
-----------------	---

Description

Size of the downloaded-repository file cache

Usage

```
repo_cache_size()
```

Value

total size of the cache in bytes (numeric). 0 when the cache is empty or absent.

See Also

[repo_cache_dir\(\)](#), [repo_cache_clear\(\)](#)

Examples

```
## Not run:  
# human-readable  
format(structure(repo_cache_size(), class = "object_size"), units = "auto")  
  
## End(Not run)
```

repo_info_cache	<i>Enable, disable, or query the repository-listing cache</i>
-----------------	---

Description

When enabled, [dryad_info\(\)](#) and [zenodo_info\(\)](#) (via [repo_check\(\)](#)'s own cache parameter) store each repository's listing result (dataset metadata + file list) on disk and reuse it on a later call for the same repository, instead of re-querying the host's API. This is a SEPARATE cache from the downloaded-file-bytes cache (see [repo_cache_dir\(\)](#)) – listing a repository and downloading its files are different costs, and a restart that has nothing new to download would otherwise still re-pay the full listing cost every time.

Usage

```
repo_info_cache(enabled = NULL)
```

Arguments

enabled	if logical, sets whether the cache is used; if NULL (default) returns the current setting
---------	---

Details

A cached listing is not refreshed automatically; a published dataset’s own metadata/file list rarely changes once public, but if a paper’s repository has genuinely been updated, clear the relevant entry with `repo_info_cache_clear()`.

Value

the current setting (logical), invisibly when setting

Examples

```
repo_info_cache()      # is the cache on?
## Not run:
repo_info_cache(TRUE)  # reuse listings across restarts

## End(Not run)
```

repo_info_cache_clear	<i>Clear the on-disk repository-listing cache</i>
-----------------------	---

Description

Deletes all cached repository listings (see `repo_info_cache()`). Safe at any time – a cleared entry is simply re-fetched from the host’s API the next time it is needed.

Usage

```
repo_info_cache_clear()
```

Value

the number of cache entries removed, invisibly

report	<i>Create a Report</i>
--------	------------------------

Description

Run specified modules on a paper and generate a report in quarto (qmd), html, or pdf format.

Usage

```
report(
  paper,
  modules = c("prereg_check", "funding_check", "coi_check", "power", "repo_check",
    "code_check", "stat_check", "stat_p_exact", "stat_p_nonsig", "stat_effect_size",
    "marginal", "ref_accuracy", "ref_replication", "ref_retraction", "ref_pubpeer",
    "ref_summary"),
  output_file = paste0(paper$paper_id, "_report.", output_format),
  output_format = c("html", "qmd"),
  report_type = c("full", "simple"),
  args = list()
)
```

Arguments

<code>paper</code>	a paper object or a paperlist object
<code>modules</code>	a vector of modules to run (names for built-in modules or paths for custom modules)
<code>output_file</code>	the name of the output file
<code>output_format</code>	the format to create the report in
<code>report_type</code>	"full" (the default) for the normal interactive report (theme toggle, sortable/paginated JavaScript tables, tabbed sections), or "simple" for a static report with none of that – plain tables, stacked headings instead of tabs, a single theme, no embedded JavaScript at all. Use "simple" for a report you plan to email: mail clients and scanners routinely flag the JavaScript a full report embeds (even though it is inert without a browser). "checklist" sorts every check by its result (to fix, to improve, ...), with what was found and what to do on separate lines; it is static too. See report_type() .
<code>args</code>	a list of arguments to pass to modules (see Details)

Details

Pass arguments to modules in a named list of lists, using the same names as the `modules` argument. You only need to specify modules with arguments.

```
args <- list(power = list(seed = 8675309))
```

Value

the module output, invisibly, with the report's file path in its `save_path` attribute (a vector of paths, one per paper, when `paper` is a paperlist)

Examples

```
## Not run:
paper <- demopaper()
report(paper)
```

```
report(paper, report_type = "simple") # email-safe, no embedded JS

## End(Not run)
```

report_app

Launch Report App

Description

Launch the Report app: upload a PDF and generate a report with one click, with privacy options for what is sent to external servers.

Usage

```
report_app(quiet = FALSE, ...)
```

Arguments

quiet	whether to show debugging messages in the console
...	arguments to pass to shiny::runApp

Value

NULL (invisibly)

Examples

```
## Not run:
report_app()

## End(Not run)
```

report_module_run

Run modules for a report

Description

Runs modules in order on the paper and orders by section and traffic light.

Usage

```
report_module_run(paper, modules, args = list())
```

Arguments

paper	a paper object
modules	a vector of modules to run
args	optional list of arguments to pass to modules

Details

Pass arguments to modules in a named list of lists, using the same names as the modules argument. You only need to specify modules with arguments.

```
args <- list(power = list(seed = 8675309))
```

Value

a list of module outputs

Examples

```
paper <- demopaper()
modules <- c("stat_p_exact", "stat_p_nonsig")
module_output <- report_module_run(paper, modules)
```

report_qmd

Create Report from Module Output

Description

Create Report from Module Output

Usage

```
report_qmd(module_output, paper = list())
```

Arguments

module_output	a list of module output (usually from report_module_run())
paper	a paper object

Value

report text

report_repository

Create a Report for a Local Repository

Description

Runs the repository modules on a folder of files on your own computer and writes a single report. Use it on a repository you have downloaded (for example with [osf_file_download\(\)](#)) to see what was shared and what could be improved, before archiving the files somewhere permanent.

Usage

```
report_repository(
  path,
  output_file = NULL,
  output_format = c("html", "qmd"),
  report_type = c("full", "simple"),
  modules = c("repo_check", "code_check", "data_check", "codebook_check"),
  args = list()
)
```

Arguments

path	path to the repository folder to check
output_file	the name of the output file. Defaults to the folder's own name with <code>_report.html</code> appended, written to the working directory. Give a path here to write it somewhere else; any folders in that path must already exist.
output_format	the format to create the report in, "html" (the default) or "qmd"
report_type	"full" (the default), "simple" (a static, email-safe report with no embedded JavaScript) or "checklist" (every check sorted by its result, also static) – see report() and report_type() .
modules	the modules to run. Defaults to the four repository modules, in the order they depend on each other. Change it to run fewer.
args	a list of extra arguments to pass to modules, named by module (see report()). <code>local_path</code> and <code>local_only</code> are set for you.

Details

Four modules run in order, each building on the one before it: `repo_check` takes an inventory of the files, `code_check` reads the analysis scripts, `data_check` reads the data files and runs data-quality checks, and `codebook_check` checks whether the data columns are documented. Only the first is told where the files are; the rest reuse its results.

Nothing is downloaded and no links are followed: only the folder you name is read. Whether a language model is used is decided by [llm_use\(\)](#), exactly as when running the modules individually.

Value

the module output, invisibly, with the report's file path in its `save_path` attribute

Examples

```
## Not run:
# check a folder you have downloaded
report_repository("how_many_registered_studies_are_published")

# write the report somewhere else
report_repository("my_study", output_file = "reports/my_study.html")

# a static, email-safe report
report_repository("my_study", report_type = "simple")

## End(Not run)
```

report_table	<i>Display a Table in a Report</i>
--------------	------------------------------------

Description

A function to display tables in reports.

Usage

```
report_table(table, colwidths = "auto", maxrows = 2, escape = FALSE)
```

Arguments

table	the data frame to show in a table, or a vector for a list
colwidths	set column widths as a vector of px (number > 1) or percent (numbers <= 1)
maxrows	if the table has more rows than this, paginate (full report) or show a scrollbar sized to approximately this many rows (simple report)
escape	whether or not to escape the table (necessary if using raw html)

Details

When `report_type()` is "simple", "simple_brief" or "checklist", this renders a plain static HTML table (via `knitr::kable()`) in a CSS-only scrollable box instead of a `DT::datatable()` widget – no JavaScript at all, so the report stays safe to email. Every row is kept and reachable by scrolling (just like the full report's paginated widget, just without the JS), except for a pathologically large table, which is truncated to `.report_table_simple_max_rows` rows with a note, so one huge table cannot make the report unboundedly large.

Value

the datatable (full report) or a `knitr::kable` object (simple report)

Examples

```
report_table(iris)
```

report_type	<i>Set or get the report type</i>
-------------	-----------------------------------

Description

Controls two independent things about how `report()/report_repository()` build a report:

- **rendering mechanics** – a full, interactive HTML report (light/dark theme toggle, sortable/paginated JavaScript tables, tabbed sections) vs. a "simple" static report with none of that – plain tables, stacked headings instead of tabs, a single theme, no inline `<script>` content. Email clients and mail scanners routinely flag the embedded JavaScript a full report carries (even though it is inert without a browser), so a "simple" report type is the setting to use before emailing a report.
- **content amount** – every module's full detail vs. only the modules flagged red/yellow/fail (i.e. what needs attention), with just their one-line summary and no expandable detail, "How It Works" callout, or validation note. This is the "brief" setting, for getting to "what to improve" as fast as possible.

Usage

```
report_type(report_type = NULL)
```

Arguments

`report_type` if one of "full", "brief", "simple", "simple_brief" or "checklist", sets the report type; NULL (the default) returns the current setting without changing it

Details

These two axes combine into four values: "full" (default), "brief", "simple", and "simple_brief". `report_table()` and any module building tabbed output (e.g. `data_check`'s per-file table tabset) check this setting themselves, since they build their markdown/HTML while a module runs – before `report()` ever selects a template.

A fifth value, "checklist", lays the report out by check instead of by module: every check of every module is sorted into to fix, to improve, decide by hand, for information, fine and not applicable, and each check that needs work says what was found and what to do on separate lines. It is rendered like "simple" (static, no JavaScript), in the colours and font of metacheck's website. A module gives its checks as a checks data frame in its output, one row per check, with the columns:

- `status`: "fix", "improve", "decide" (decide by hand), "info", "fine" or "na" (not applicable)
- `title`: the name of the check
- `found`: what was found (markdown)

- todo: what to do (markdown)
- and optionally why and note (markdown), section (a label shown next to the title), and details (markdown, for example a `scroll_table()` with `details_title` (the line that unfolds it))

A module without checks is shown as one check: its traffic light gives the result (red: to fix, yellow: to improve, green: fine), its summary text what was found, and its report the folded details. A module can also give intro (a paragraph shown at the top) and logo (the path of an image shown next to metacheck's logo).

Value

the current option value

Examples

```
report_type()
report_type("simple")
report_type("brief")
report_type("simple_brief")
report_type("checklist")
report_type("full")
```

repro_defined_vars	<i>Find the variables each script defines at top level</i>
--------------------	--

Description

When a script errors with object 'X' not found, the fix is usually to run the script that *defines* X first. This scans each file for **top-level** variable assignments — `x <- ...`, `x = ...`, `x <<- ...`, and `assign("x", ...)` — so an undefined-variable error can be matched to the file that would supply it (see the module's corrective re-run). Only assignments at the start of a (comment-free) line are taken, which approximates "top level": an assignment indented inside a function/for body is not a global the next sourced script would see, and taking it would create false ordering edges.

Usage

```
repro_defined_vars(code_text_list)
```

Arguments

`code_text_list` a named list of code-text character vectors, one per file (names are the `file_names`)

Value

a data frame with `file_name` and a list-column `defines` (character vector of variable names each file assigns at top level).

repro_dependencies	<i>Collect the package dependencies a set of code files declare</i>
--------------------	---

Description

Walks each code file with `code_library_names()` and returns the distinct packages the paper's code loads, tagged with how each was referenced and whether it is installable from CRAN, from a named GitHub/URL source, or is a base/recommended package already shipped with R (so a run need not install it). This is the input list a run would install; it is deliberately name-only, because static analysis cannot know which *version* was used — running against the latest packages and observing a break is itself a finding (it argues the authors should have pinned versions).

Usage

```
repro_dependencies(code_text, lang = "R")
```

Arguments

<code>code_text</code>	the code text for a single file (character vector), OR a list of such vectors (one per file) to pool across files
<code>lang</code>	the language (only R declares installable packages here)

Details

GitHub/URL sources are read from the code text, not the package name: a call `remotes::install_github("user/repo")` names the source `user/repo`, which is matched back to the package by the repo's basename. When a script pins a ref (`user/repo@v1.2`), the ref is kept.

Value

a data frame with columns `package`, `source` (`cran`, `bioc`, `github`, `url`, or `base`), `ref` (the source path/ref for `github/url`, else `NA`), and `base` (logical, `TRUE` for a base/recommended package). One row per distinct package. Empty frame (same columns) when none are found.

Examples

```
code_text <- c(
  "library(dplyr)",
  "remotes::install_github('tidyverse/ggplot2@v3.4.0')",
  "x <- stats::sd(1:10)"
)
repro_dependencies(code_text)
```

repro_dependencies_py *Collect the package dependencies a set of Python files declare*

Description

The Python sibling of `repro_dependencies()`: walks each file's `import/ from ... import` statements via `code_library_names()` (`lang = "Python"`), and additionally reads any `requirements.txt/pyproject.toml` the repository supplies for version pins — static analysis of the code alone cannot see those (a script that does `import numpy` says nothing about which version), so when a manifest is present its pin is attached to the matching imported package instead of being reported separately. A top-level package name is tagged `base` (the Python standard library; a run never needs to `pip install` these) or `pypi` — unlike R's CRAN/ Bioconductor/GitHub/URL split, this does not attempt to tell a PyPI install apart from a `pip install git+.../local-path` install named in a `requirements.txt` line: both are reported `source = "pypi"`, with `ref` carrying the raw requirement line verbatim when it is not a bare `package==version` pin, so the install step can still attempt it.

Usage

```
repro_dependencies_py(code_text, manifest_text = NULL)
```

Arguments

<code>code_text</code>	the code text for a single file (character vector), OR a list of such vectors (one per file) to pool across files
<code>manifest_text</code>	optional character vector: the text of a <code>requirements.txt</code> (one requirement per line) or the <code>[project]/[tool.poetry]</code> dependency lines of a <code>pyproject.toml</code> , if the repository has one. <code>NULL</code> (default) when none was found — dependencies are then name-only, exactly like a code-only R scan.

Value

a data frame with columns `package`, `source` (`pypi` or `base`), `ref` (the manifest's own pin/requirement text for that package, or `NA` when the package was seen only via `import`), and `base` (logical, `TRUE` for a standard-library module). One row per distinct package. Empty frame (same columns) when none are found.

Examples

```
code_text <- c("import numpy as np", "from sklearn.linear_model import LinearRegression")
repro_dependencies_py(code_text)
```

```
repro_docker_available
```

Is Docker available and running?

Description

Checks `docker info` succeeds, so a caller can fail early and clearly (before materialising anything) rather than partway through a batch of papers with a cryptic `docker run` error. Used as the guard for `sandbox = "docker"` the same way `requireNamespace("callr", ...)` guards the process backend.

Usage

```
repro_docker_available()
```

Value

a list `ok` (logical) and `msg` (character, empty on success — otherwise a human-readable reason, e.g. Docker not installed or the daemon not running)

```
repro_file_io
```

Extract the files a script reads, writes, and sources

Description

A thin wrapper that runs the static reference scans over each code file and returns the per-file read / write / source basenames `repro_run_order()` needs. Reads/writes come from `code_file_refs()` split by call type; sources from a scan for `source()` (R only — see `lang` below). All returned as lowercased basenames.

Usage

```
repro_file_io(code_text_list, lang = "R")
```

Arguments

<code>code_text_list</code>	a named list of code-text character vectors, one per file (names are the <code>file_names</code>), OR a named list of such lists when files are in more than one language (see <code>lang</code>)
<code>lang</code>	the language each element of <code>code_text_list</code> is in: either a single string applied to every file (default <code>"R"</code>), or a named character vector keyed by <code>file_name</code> for a mixed-language set. Python has no <code>source()</code> -equivalent for reusing a SIBLING analysis script by relative path (its code-reuse mechanism, <code>import</code> , names an installed module, not a local file <code>run_order</code> must sequence), so sources is always empty for a Python file; <code>open(..., "w"/"a"/...)</code> is classified as a write only when that same line's own mode argument says so (a bare <code>open(x)</code> defaults to Python's own read mode and is otherwise ambiguous between the two).

Value

a data frame with `file_name` and `list-columns` reads, writes, sources (each a character vector of basenames).

<code>repro_install_deps</code>	<i>Install a paper's declared dependencies into a throwaway library</i>
---------------------------------	---

Description

Installs each non-base dependency into `lib_dir` (a temp library, never the user's), so a run does not mutate the user's installed packages. CRAN packages come from `install.packages()`; Bioconductor packages (`source == "bioc"`, see `repro_dependencies()/repro_bioc_packages()`) from `BiocManager::install()`, installing `BiocManager` itself first if absent; GitHub/URL sources from the ref the code named (`remotes::install_github()` / `install.packages(url)`), honouring a pinned `@ref` when present. Every install's outcome is recorded; installing runs package build/configure scripts, so this is part of the gated execute phase.

Usage

```
repro_install_deps(install_deps, lib_dir, cran_to_main_lib = FALSE)
```

Arguments

`install_deps` the module's `install_deps` frame (package, source, ref), base packages already excluded

`lib_dir` the throwaway library path (created if absent); ignored for CRAN-source packages when `cran_to_main_lib = TRUE` (see below)

`cran_to_main_lib`
 if TRUE, CRAN-source packages are installed into your DEFAULT library (no `lib` argument — same as calling `install.packages(pkg)` yourself) instead of the throwaway `lib_dir`, and so PERSIST after the run rather than being deleted with the sandbox. This trades a one-time, permanent change to your R library for install work that is shared across every later call needing the same package (useful when `reproducibility_check()` runs across many papers in one session/script: a CRAN package installed for an early paper is already present, and skipped as already-installed, for every later one). GitHub/URL sources are deliberately excluded from this and always go into the throwaway `lib_dir` regardless of this argument — those are arbitrary, unreviewed code you likely do not want permanently in your main library. Default FALSE (everything throw-away, the original behaviour).

A CRAN-source package that `install.packages()` cannot find is not necessarily a broken dependency: CRAN removes (archives) packages that stop passing its checks, but every version ever published stays available as a source tarball under the CRAN Archive (<https://cran.r-project.org/src/contrib/Archive/<pkg>/>). When the live-CRAN install fails, this retries once from the Archive's most recent version before giving up — a package the paper's authors could install at

the time (and that later dropped off live CRAN) still installs, instead of the run recording an "errored" script for a cause that has nothing to do with the paper's code. The retry is recorded in `via_archive` so the report can say the run used a stale-but-real version rather than the one the authors had.

Value

a data frame with `package`, `source`, `installed` (logical), `message` (error text on failure, else ""), `via_archive` (logical, TRUE when installed from the CRAN Archive after a live-CRAN failure), and `category` (see `.repro_classify_install_message()`; NA for a successful install).

repro_install_deps_docker

Install R dependencies inside a throwaway Docker container

Description

The Docker-backed twin of `repro_install_deps()`: same return contract, same throwaway-library semantics, but the actual `install.packages()/remotes::install_github()` call runs INSIDE a container (default networking, since CRAN/GitHub must be reachable), writing into `lib_dir` via a bind mount — so the run phase (`repro_run_scripts_docker()`), which mounts the same `lib_dir` with the network off, finds the packages already there.

Usage

```
repro_install_deps_docker(
  install_deps,
  lib_dir,
  image = "rocker/r-ver:latest",
  timeout = 600
)
```

Arguments

<code>install_deps</code>	a data frame as produced inside <code>reproducibility_check()</code> (<code>package</code> , <code>source</code> , <code>ref</code> , <code>base</code>) — the non-base rows of <code>repro_dependencies()</code> 's output
<code>lib_dir</code>	throwaway library directory ON THE HOST (created if absent); bind-mounted into the container at <code>/rlib</code>
<code>image</code>	the Docker image to install into (see <code>.repro_docker_image_for()</code>)
<code>timeout</code>	per-package-batch timeout in seconds for the whole install container run (default 600 — installs can be slow, esp. compiling from source)

Details

Unlike `repro_install_deps()`, there is no `cran_to_main_lib` option: a container has no access to (and must not be given access to) the host's real R library, so every install always goes into the throwaway `lib_dir`.

Value

a data frame with package, source, installed (logical), message (error text on failure, else ""), via_archive (always FALSE — the CRAN Archive retry [repro_install_deps\(\)](#) does is not duplicated here; a package unavailable on live CRAN is simply recorded as failed), and category (see [.repro_classify_install_message\(\)](#); NA for a successful install) — same columns as [repro_install_deps\(\)](#) so callers do not need to branch on which backend ran.

repro_install_deps_py *Install a paper's declared Python dependencies into a throwaway environment*

Description

The process-sandbox twin of [repro_install_deps\(\)](#): installs each non-stdlib dependency via `pip install --target <lib_dir>`, so a run does not touch the host's own site-packages. A manifest-pinned requirement (ref non-NA — see [repro_dependencies_py\(\)](#)) is installed by that exact requirement text (honouring the pin); an import-only package (ref NA) is installed by bare name, latest version. Installing runs a package's own build/setup code, so this is part of the gated execute phase, same as its R counterpart.

Usage

```
repro_install_deps_py(
  install_deps,
  lib_dir,
  python_bin = "python",
  timeout = 300
)
```

Arguments

install_deps	the module's Python install_deps frame (package, source, ref), base/stdlib rows already excluded
lib_dir	the throwaway install target (created if absent); passed to <code>pip install --target</code>
python_bin	the python executable to invoke <code>-m pip</code> with (default "python", resolved via <code>Sys.which()</code> at call time so a missing interpreter fails with a clear message rather than a cryptic pip error)
timeout	per-pip install timeout in seconds (default 300)

Value

a data frame with package, source, installed (logical), message (error text/pip stderr on failure, else ""), via_archive (always FALSE — pip has no CRAN-Archive-style fallback registry to retry against), and category (always NA — pip's own error text is not classified into categories the way [.repro_classify_install_message\(\)](#) does for R's `install.packages()/BiocManager` error shapes) — same column SET as [repro_install_deps\(\)](#) so callers do not need to branch on which language ran.

repro_install_deps_py_docker

Install Python dependencies inside a throwaway Docker container

Description

The Docker-backed twin of `repro_install_deps_py()`: same return contract, same throwaway-library semantics, but the actual `pip install` runs **INSIDE** a container (default networking, since PyPI must be reachable), writing into `lib_dir` via a bind mount — so the run phase (`repro_run_scripts_py_docker()`), which mounts the same `lib_dir` with the network off, finds the packages already there.

Usage

```
repro_install_deps_py_docker(
    install_deps,
    lib_dir,
    image = .repro_docker_default_image_py,
    timeout = 600
)
```

Arguments

<code>install_deps</code>	a data frame as produced inside <code>reproducibility_check()</code> (package, source, ref, base) — the non-base rows of <code>repro_dependencies_py()</code> 's output
<code>lib_dir</code>	throwaway <code>pip install --target</code> directory ON THE HOST (created if absent); bind-mounted into the container at <code>/pylib</code>
<code>image</code>	the Docker image to install into (see <code>.repro_docker_default_image_py</code>)
<code>timeout</code>	timeout in seconds for the whole install container run (default 600)

Value

a data frame with package, source, installed (logical), message (pip's stderr on failure, else ""), `via_archive` (always FALSE — pip has no CRAN-Archive-style fallback), and category (always NA) — same columns as `repro_install_deps_py()` so callers do not need to branch on which backend ran.

repro_materialize_layout

Materialise the Psych-DS layout for a paper into a throwaway directory

Description

The execute phase runs the paper's code against the Psych-DS layout, but the module has only the *plan* (psychds_check's file → target_path) and the current on-disk location of each source file (data_check's structure). This builds the tree the plan describes inside a fresh temp directory: for each planned file with a real target and an available source copy, the source is copied to <root>/<target_path>. Nothing outside root is written, and root is the caller's to delete.

Usage

```
repro_materialize_layout(plan, structure_df, root)
```

Arguments

plan	the psychds_check table (file_name, target_path), or the package layout's plan (.repro_package_plan()), whose source column names each row's own file
structure_df	the data_check structure table (file_name, file_location)
root	the temp directory to build into (created if absent)

Value

root, invisibly, with attribute "materialised": a data frame of target_path / source / ok (whether the copy succeeded) rows.

repro_missing_inputs	<i>Diagnose why a referenced input file is unavailable</i>
----------------------	--

Description

When a script reads a file that is not present, the reason matters: a file deliberately withheld because it was too large to download is a different finding from a file simply absent from the repository. This cross-references the referenced basenames against the file plan and the download-skip records download_repo_files() attaches, and classifies each unresolved reference.

Usage

```
repro_missing_inputs(refs, plan, structure_df, skipped = NULL)
```

Arguments

refs	a character vector of referenced file basenames (unresolved reads)
plan	the psychds_check table (file_name + target_path)
structure_df	the data_check structure table (file_name + file_location + file_size)
skipped	optional data frame of size-skipped files (the oversize_skipped / gated attributes of download_repo_files() , with a file_name column)

Value

a data frame with `basename`, `status` (one of `in_repo_not_downloaded`, `withheld_size`, `in_archive` (the file is in the repository only inside a .zip or .tar, so the code cannot read it as the package is), `absent`), `detail` (a human-readable reason), and `similar_to` (a candidate near-match `basename` from the repository's file list, or NA — only ever set for absent rows; see below).

<code>repro_rewrite_paths</code>	<i>Rewrite a code file's data-file paths to the Psych-DS layout</i>
----------------------------------	---

Description

A script reads and writes data by relative path (`read_csv("raw/x.csv")`, `saveRDS(m, "../out/model.rds")`). When the release is re-laid-out to Psych-DS (data under `data/`, per-study `study-<group>/data/`), those paths no lives, using the file→target plan `psychds_check` produced.

Usage

```
repro_rewrite_paths(
  code_text,
  file_name,
  plan,
  lang = "R",
  structure_df = NULL
)
```

Arguments

<code>code_text</code>	the code text for a single file (character vector)
<code>file_name</code>	the script's own name/path (used to derive its study group)
<code>plan</code>	the <code>psychds_check</code> table: one row per file with <code>file_name</code> , <code>target_path</code> , <code>current_path</code> , and (for converted tabular files) <code>original_target</code>
<code>lang</code>	the language: "R" or "Python" are rewritten; any other value returns the empty frame. Python's format-string call forms (f-strings, <code>.format()</code> , <code>os.path.join()</code>) are NOT resolved to a call row the way R's <code>sprintf()/paste()/file.path()</code> are (see <code>call_refs/</code> repro_format_call_refs() below) — only a literal quoted path is rewritten for Python, so a path built by string concatenation/formatting is left alone and reported <code>matched = FALSE</code> rather than guessed at.
<code>structure_df</code>	optional <code>data_check</code> structure table (<code>file_name</code> , <code>file_location</code>) — when supplied, enables the content-hash mirror collapsing described above. NULL (default) skips it.

Details

Matching is by **basename** (a script's `../data/x.csv` and the repo's `raw/x.csv` are the same file seen from different working directories, so the prefix is ignored — the same choice `code_file_refs()` makes). When several plan files share a basename (a `demographics.csv` in more than one study), the ambiguity is resolved by **study group**: the candidate whose target path is in the same study as the script (derived from the paths via `.data_group_from_path()`) is chosen. Only when that still leaves more than one candidate is the reference left unrewritten and flagged.

A plan row for a converted tabular file (psychds_check's `convert/` `original_target` columns — a `.dta/.sav/.xlsx/...` source, whose `target_path` is the re-exported `_data.csv`, with the untouched original kept alongside at `original_target`) is matched **by the reference's own extension**: a reference ending in the original's extension (`read_dta("math.dta")`) is rewritten to `original_target`, since a format-specific reader needs the real file, not the CSV re-export — using `target_path` there would silently point the reader at bytes it cannot correctly parse. A reference that already names the CSV form (`"math_data.csv"`, or a bare `.csv` extension) still rewrites to `target_path` as before.

A path assembled at runtime with `sprintf()/paste()/paste0()/file.path()` (see `.repro_format_call_refs()`) is matched the same way, using the format string's own literal basename (with any resolvable placeholder substituted via `.repro_simple_string_vars()`) — but since the true "reference" there is the WHOLE call, not the quoted substring inside it, such rows carry `is_call = TRUE` and `ref` holds the full call text (`sprintf("%s/%s/x.csv", wd, wd_data)`), so `repro_write_scripts()` replaces the entire expression rather than corrupting it with a substring swap that would orphan the call's trailing arguments. A call whose placeholders could not all be resolved is reported unmatched (`matched = FALSE`) rather than guessed, and is separately flagged in the report as fragile authoring practice regardless of whether it resolved.

A basename can also collide across plan rows for a reason that is NOT real ambiguity at all: a paper linking several OSF components that mirror each other (see `repro_materialize_layout()`'s own de-duplication for the execution-time counterpart of this) commonly lists the identical physical file more than once, and the study-grouping step can assign DIFFERENT (and for all-but-one, WRONG) groups to different mirrors of the same file — e.g. one mirror's folder structure carries a "Study 1" hint the others lack, so only that one gets grouped correctly. When `structure_df` is supplied, candidates are first collapsed by the CONTENT HASH of their resolved source files: byte-identical candidates count as ONE, using whichever one's own group agrees with the majority (ties broken by the first occurrence) — confirmed against a real corpus paper where 4 mirrors of the same `Anonymized_2024-08-23.csv` were grouped `ex1/ex1/ex1/ex3`, which made the existing "do all candidates already agree on one target" shortcut fail (3 way agree, 1 disagrees) and fell through to `ambiguous = TRUE` even though there was only ever one real file. Without `structure_df` (or when a candidate's source cannot be resolved/read), this collapsing step is skipped and candidates are compared as before.

Value

a data frame with one row per referenced path, columns `ref` (the path as written, or the full call text when `is_call`), `basename`, `matched` (logical, a plan file was found), `target` (its Psych-DS path, or NA), `ambiguous` (logical, several plan files matched and the study could not disambiguate), `n_candidates`, and `is_call` (logical, TRUE when `ref` is a whole `sprintf()/paste()`-family call rather than a literal path — see above). Empty frame when the script references no files.

Examples

```
plan <- data.frame(
  file_name = c("demographics.csv", "scores.csv"),
  target_path = c("study-ex1/data/study-demographics_data.csv",
                  "study-ex1/data/study-scores_data.csv"),
  current_path = c("ex1/demographics.csv", "ex1/scores.csv")
)
code_text <- c('d <- read.csv("data/demographics.csv")')
repro_rewrite_paths(code_text, "ex1/analysis.R", plan)
```

repro_run_order	<i>Determine the order code files must run in</i>
-----------------	---

Description

Scripts in a repository usually depend on each other: a preparation script writes a file a later analysis script reads, or one source()s another. Running them in the wrong order fails for reasons that have nothing to do with reproducibility, so a run must order them first. This derives an order from three signals, in decreasing strength:

Usage

```
repro_run_order(files, extra_edges = NULL)
```

Arguments

files	a data frame with one row per code file, columns file_name, and reads / writes / sources list-columns of basenames (as produced by repro_file_io()). When reads/writes/sources are absent they are derived from a code_text list-column if present.
-------	--

Details

1. **read-after-write** — if file A's code writes x.csv and file B's code reads x.csv, A must precede B (hard data dependency);
2. **source()** **edges** — if A source()s B, B must precede A;
3. **numeric filename prefixes** — 0_prep.R before 1_analysis.R (a weak tie-breaker, not a dependency).

The read/write and source references come from the shared static analysis ([code_file_refs\(\)](#) and a source scan); matching is by basename (as elsewhere). The dependency graph is topologically sorted; a cycle (A reads a file B writes and vice versa) cannot be ordered and is reported.

Value

a data frame with columns `file_name`, `order` (1-based run position, NA when unplaceable), `depends_on` (comma-joined `file_names` that must run earlier), and `order_basis` (dependency, numeric, or none). Attribute `"cycle"` is a character vector of `file_names` in a dependency cycle (empty when acyclic); attribute `"ambiguous"` is TRUE when no ordering signal placed a multi-file set (they share `order = NA`, basis none). Attribute `"fuzzy_sources"` is a data frame (from, to) of `source()` edges that only resolved after normalising both basenames (a renumbered/ reworded script reference — see `.repro_normalize_basename()`), empty when every `source()` reference matched an exact basename. Attribute `"source_edges"` is a data frame (from, to) of every resolved `source()` edge, exact or fuzzy: from is the sourced file, to the file that sources it.

Examples

```
files <- data.frame(file_name = c("1_analysis.R", "0_prep.R"))
files$reads <- list("clean.csv", character(0))
files$writes <- list(character(0), "clean.csv")
files$sources <- list(character(0), character(0))
repro_run_order(files)
```

<code>repro_run_scripts</code>	<i>Run the paper's scripts, in order, in isolated subprocesses</i>
--------------------------------	--

Description

Runs each script with `callr::r()` from `run_dir`, with `lib_dir` prepended to the subprocess `.libPaths()` so the throwaway library is seen. Each run is bounded by `timeout` seconds; its exit, error, stdout and stderr are captured. A script whose inputs are known-missing (in `skip`) is not run — it is recorded `skipped_missing_inputs`, since it cannot succeed and would only produce a confusing error. `callr` isolates a crash to the subprocess; it does not sandbox the filesystem or network, so this only runs on explicit opt-in.

Usage

```
repro_run_scripts(
  run_tbl,
  order,
  lib_dir = NULL,
  timeout = 600,
  skip = character(0),
  parses = NULL,
  failed_deps = character(0),
  deadline = NULL
)
```

Arguments

run_tbl	repro_write_scripts() output (file_name, script_path, run_dir)
order	a vector of file_names in the order to run (unplaceable files are appended at the end in input order)
lib_dir	throwaway library to expose to each subprocess (or NULL)
timeout	per-script timeout in seconds
skip	character vector of file_names to record as skipped_missing_inputs instead of running
parses	named logical (by file_name); a file that will not parse is recorded not_parsed and not run
failed_deps	character vector of package names that repro_install_deps() could not install (CRAN Archive retry included) — used only to tell apart a script that failed because ITS OWN dependency is genuinely unavailable (dependency_unavailable) from a script that failed for any other reason (errored), so an unavailable package does not read as a bug in the paper’s code
deadline	the time (Sys.time() plus the time for all scripts together) after which no script is started, recorded as skipped_time_limit; a running script is stopped then. NULL (the default) sets no such limit.

Value

a data frame with file_name, outcome (one of ran_ok, errored, timed_out, not_parsed, skipped_missing_inputs, dependency_unavailable), error (message or ""), error_type (undefined_variable, timeout, dependency_unavailable, runtime, or NA), undefined_var (the missing variable name when error_type == "undefined_variable", else NA), stdout, stderr, elapsed (seconds), and script_lines (list-column: the EXECUTED script’s own text, one element per source line — the exact text stdout’s echoed statements were run from, so read_r_output() can match them back to a line number; empty character(0) for a script that did not run).

repro_run_scripts_docker

Run the paper’s scripts, in order, each in an isolated Docker container

Description

The Docker-backed twin of repro_run_scripts(): same return contract, same per-script semantics (skip on missing inputs / parse failure, timeout, undefined-variable / dependency-unavailable classification, captured result objects), but each script runs via docker run --network none --read-only --user <non-ro instead of a callr subprocess — so the code cannot reach the network or touch anything outside the mounted sandbox and library, even deliberately. See this file’s header comment for the two-phase (install-then-run) design and why the library is mounted at /rlib, not /lib.

Usage

```

repro_run_scripts_docker(
  run_tbl,
  order,
  sandbox_root,
  lib_dir = NULL,
  image = "rocker/r-ver:latest",
  timeout = 600,
  skip = character(),
  parses = NULL,
  failed_deps = character(),
  deadline = NULL
)

```

Arguments

run_tbl	repro_write_scripts() output (file_name, script_path, run_dir) — run_dir must be the SAME directory as (or a subdirectory of) sandbox_root, so it can be bind-mounted as a whole
order	a vector of file_names in the order to run
sandbox_root	the materialised layout root on the HOST (from repro_materialize_layout()) — mounted read-write at /sandbox
lib_dir	throwaway library ON THE HOST (or NULL) — mounted read-only at /rlib when supplied
image	the Docker image to run scripts in (see .repro_docker_image_for())
timeout	per-script timeout in seconds
skip	character vector of file_names to record as skipped_missing_inputs instead of running
parses	named logical (by file_name); a file that will not parse is recorded not_parsed and not run
failed_deps	character vector of package names repro_install_deps_docker() could not install
deadline	the time (Sys.time() plus the time for all scripts together) after which no script is started, recorded as skipped_time_limit; a running script is stopped then. NULL (the default) sets no such limit.

Value

a data frame — identical columns to [repro_run_scripts\(\)](#)'s return: file_name, outcome, error, error_type, undefined_var, stdout, stderr, elapsed, script_lines, captures

repro_run_scripts_py *Run Python scripts, in order, each in an isolated subprocess*

Description

The Python sibling of `repro_run_scripts()`: runs each script `WHOLE`, as a python `<script>` subprocess via `processx::run()` (not `callr`, which is R-specific), with the working directory set to the materialised sandbox. Unlike `repro_run_scripts()`'s `callr`-based runner, there is no per-statement capture of result objects here (see this file's header for why) — only the whole script's stdout/stderr/exit status/elapsed time are recorded, so captures is always `NULL` for every row (kept as a column, not dropped, so the module's downstream code that expects the same shape `repro_run_scripts()` returns does not need to branch by language).

Usage

```
repro_run_scripts_py(
  run_tbl,
  order,
  lib_dir = NULL,
  python_bin = "python",
  timeout = 600,
  skip = character(0),
  parses = NULL,
  failed_deps = character(0),
  deadline = NULL
)
```

Arguments

<code>run_tbl</code>	<code>repro_write_scripts()</code> output (<code>file_name</code> , <code>script_path</code> , <code>run_dir</code>)
<code>order</code>	a vector of <code>file_names</code> in the order to run
<code>lib_dir</code>	throwaway <code>pip install --target</code> directory (or <code>NULL</code>) — prepended to <code>PYTHONPATH</code> so an installed dependency is importable
<code>python_bin</code>	the python executable to run scripts with (default "python")
<code>timeout</code>	per-script timeout in seconds
<code>skip</code>	character vector of <code>file_names</code> to record as <code>skipped_missing_inputs</code> instead of running
<code>parses</code>	named logical (by <code>file_name</code>); a file that will not parse (see <code>code_parse_py()</code> /the module's own syntax check) is recorded <code>not_parsed</code> and not run
<code>failed_deps</code>	character vector of package names <code>repro_install_deps_py()</code> could not install
<code>deadline</code>	the time (<code>Sys.time()</code> plus the time for all scripts together) after which no script is started, recorded as <code>skipped_time_limit</code> ; a running script is stopped then. <code>NULL</code> (the default) sets no such limit.

Details

This isolates a CRASH, not the filesystem or network — the same caveat `repro_run_scripts()`'s own roxygen states for its callr backend applies identically here: the script can read/write/delete anywhere this process can, and reach the network freely. Use `sandbox = "docker"` (`repro_run_scripts_py_docker()`) for code you do not trust.

Value

a data frame — same columns as `repro_run_scripts()`'s return: `file_name`, `outcome`, `error`, `error_type`, `undefined_var`, `stdout`, `stderr`, `elapsed`, `script_lines`, `captures` (always NULL per row — see above)

```
repro_run_scripts_py_docker
```

Run Python scripts, in order, each in an isolated Docker container

Description

The Docker-backed twin of `repro_run_scripts_py()`: same return contract, same per-script semantics (skip on missing inputs / parse failure, timeout, error classification), but each script runs via `docker run --network none --read-only --user <non-root>` instead of a `processx` subprocess on the host — so the code cannot reach the network or touch anything outside the mounted sandbox and library, even deliberately.

Usage

```
repro_run_scripts_py_docker(
  run_tbl,
  order,
  sandbox_root,
  lib_dir = NULL,
  image = .repro_docker_default_image_py,
  timeout = 600,
  skip = character(0),
  parses = NULL,
  failed_deps = character(0),
  deadline = NULL
)
```

Arguments

<code>run_tbl</code>	<code>repro_write_scripts()</code> output (<code>file_name</code> , <code>script_path</code> , <code>run_dir</code>) — <code>run_dir</code> must be the SAME directory as (or a subdirectory of) <code>sandbox_root</code>
<code>order</code>	a vector of <code>file_names</code> in the order to run
<code>sandbox_root</code>	the materialised layout root on the HOST — mounted read-write at <code>/sandbox</code>

lib_dir	throwaway pip install --target library ON THE HOST (or NULL) — mounted read-only at /pylib when supplied
image	the Docker image to run scripts in (see .repro_docker_default_image_py)
timeout	per-script timeout in seconds
skip	character vector of file_names to record as skipped_missing_inputs instead of running
parses	named logical (by file_name); a file that will not parse is recorded not_parsed and not run
failed_deps	character vector of package names repro_install_deps_py_docker() could not install
deadline	the time (Sys.time() plus the time for all scripts together) after which no script is started, recorded as skipped_time_limit; a running script is stopped then. NULL (the default) sets no such limit.

Value

a data frame — identical columns to [repro_run_scripts_py\(\)](#)'s return: file_name, outcome, error, error_type, undefined_var, stdout, stderr, elapsed, script_lines, captures (always NULL)

repro_write_scripts	<i>Write path-rewritten scripts into the materialised layout</i>
---------------------	--

Description

Each script's referenced data paths are rewritten to their Psych-DS target (via the per-file [repro_rewrite_paths\(\)](#) result) so that, run from root, the reads/writes resolve against the materialised tree. The rewritten script text is written to root/<script target_path> (or, when the plan has no target for the script itself, to root/<basename> at the top level so it is still runnable). A reference left ambiguous or unmatched is written unchanged — the static report already flagged it.

Usage

```
repro_write_scripts(
  code_text_list,
  rewrite_list,
  plan,
  root,
  inject_libs = NULL,
  lang = "R",
  script_paths = NULL,
  writes_in_place = FALSE
)
```

Arguments

<code>code_text_list</code>	named list of per-file code text (names are <code>file_names</code>)
<code>rewrite_list</code>	named list of <code>repro_rewrite_paths()</code> results, same order
<code>plan</code>	the <code>psychds_check</code> table (to find each script's own <code>target_path</code>)
<code>root</code>	the materialised layout root
<code>inject_libs</code>	optional named character vector/list (names are <code>file_names</code> already present in <code>code_text_list</code>) — one or more package names to inject as <code>library(<pkg>)</code> calls at the very top of that script's text before it is written. Used by the module's corrective re-run for a could not find function "X" error (see .repro_missing_fn_pkgs() for which packages) — reusing this function's existing text-rewrite plumbing rather than a separate rewrite path. NULL (default) injects nothing, unchanged from before this parameter existed. R only: there is no Python corrective re-run (the module's missing-dependency/undefined-name recovery is namespace-export-based — see .repro_find_export_pkg() — with no equivalent built for Python's import statements), so this is ignored for a file whose <code>lang</code> is "Python".
<code>lang</code>	the language each element of <code>code_text_list</code> is in: a single string applied to every file (default "R"), or a named character vector keyed by <code>file_name</code> for a mixed-language set (same shape as repro_file_io() 's own <code>lang</code>). Selects the write-redirect function names (.repro_redirect_writes()) and the working-directory override that is replaced with a statement that does nothing (<code>setwd()</code> becomes <code>invisible(NULL)</code> in R, <code>os.chdir()</code> becomes <code>pass</code> in Python). Python's font-substitution analogue is deliberately NOT built: an unavailable font is a matplotlib WARNING with a silent DejaVu Sans fallback, not the hard "invalid font type" failure family <code>=/base_family =</code> causes in R's graphics devices (the actual reason this transform exists for R) — rewriting Python's many <code>rcParams/plt.rc()/fontdict=</code> spellings would be solving a failure mode Python's own ecosystem does not have here.
<code>script_paths</code>	optional character vector, one per element of <code>code_text_list</code> : the path (relative to <code>root</code>) to write that script to, as the package layout (.repro_package_plan()) needs. NULL (default), or an NA element, places the script by its basename in <code>plan</code> , as before.
<code>writes_in_place</code>	if TRUE (the package layout), a write stays where the code puts it, so a later script finds the file where it expects it; only a write to an absolute path or out of the package is redirected to <code>output/</code> . FALSE (default) redirects every write, as before.

Details

A `setwd()` call in analysis code is bad practice: it hardcodes an assumption about where the code runs (often an absolute path on the author's own machine, e.g. `setwd("D:/Dropbox/...")`), and it overrides the working directory the runner sets to the materialised layout — so the script's relative reads then resolve against a directory that does not exist here, and the reproduction fails for a reason that has nothing to do with the analysis. To run the code we therefore **comment out every** `setwd()` call (recorded per file in `setwd_removed` and flagged in the report). The transform is in the same

spirit as the path rewriting: it makes the author's script runnable in its new location without changing the analysis. A script that only reproduces after a `setwd()` is removed is reported as "reproducible after ignoring a `setwd()` that should not have been in the code and needs to be fixed".

Value

a data frame with `file_name`, `script_path` (absolute path written), `run_dir` (the working directory a run should use — always root), `setwd_removed` (count of `setwd()/os.chdir()` calls replaced with a statement that does nothing), `setwd_paths` (comma-joined paths those calls named, for the warning), `family_replaced` (count of `family =/base_family =` named-font arguments replaced with "sans", so a plot call does not fail with "invalid font type" for a font not registered on the machine running the sandbox), `family_detail` (semicolon-joined `<old> -> "sans"` text for each replacement), and `library_injected` (the package name(s) injected via `inject_libs` for that file, comma-joined, or NA when none was).

researchdata4tu_file_download

Download all files from a 4TU.ResearchData article

Description

Thin wrapper around `figshare_file_download()` with `host = "data.4tu.nl"` — see the note at the top of this file. Creates a directory for the article and downloads all of its files. Returns (invisibly) a data frame with file info.

Usage

```
researchdata4tu_file_download(
  researchdata4tu_id,
  download_to = ".",
  max_file_size = 10,
  max_download_size = 100,
  unzip_types = NULL,
  pb = NULL
)
```

Arguments

<code>researchdata4tu_id</code>	a 4TU.ResearchData article ID, uuid, URL, or DOI
<code>download_to</code>	path to download to
<code>max_file_size</code>	maximum file size to download (in MB) - set to NULL or Inf for no restrictions
<code>max_download_size</code>	maximum total size to download - set to NULL or Inf for no restrictions

unzip_types	file categories to extract from .zip files in the article rather than downloading the zip whole, named as <code>data_classify_files()</code> names them: "data", "code", "materials", "documentation", "output", "unknown". More than one may be given. NULL (the default) downloads zips whole and leaves them zipped, as before. <code>max_file_size</code> applies to each extracted member rather than to the archive.
pb	a progress bar passed from another function

Details

Uses `researchdata4tu_pat()`, not `figshare_pat()`, for authentication (see that function's documentation for why the two are separate).

Value

data frame of file info. When members are extracted from a zip, its row reports the zip with extracted giving the number of members written; `size_on_disk` and `checksum_ok` are then NA, because what is on disk is the members rather than the archive 4TU.ResearchData published a size and MD5 for.

Examples

```
## Not run:
researchdata4tu_file_download("16766929")

# take only the data files out of any zip in the article
researchdata4tu_file_download("16766929", unzip_types = "data")

## End(Not run)
```

researchdata4tu_info *Retrieve info from 4TU.ResearchData by URL*

Description

Thin wrapper around `figshare_info()` with `host = "data.4tu.nl"` – see the note at the top of this file for why 4TU.ResearchData's Djehuty platform can reuse the Figshare-family implementation directly.

Usage

```
researchdata4tu_info(researchdata4tu_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

researchdata4tu_url	a 4TU.ResearchData URL or DOI, or a table containing them (e.g., as created by <code>researchdata4tu_links()</code>)
id_col	the index or name of the column that contains 4TU.ResearchData URLs, if <code>researchdata4tu_url</code> is a table
pb	a progress bar passed from another function
cache	if TRUE, reuse a previously cached listing for an article already looked up (see <code>repo_info_cache()</code>) instead of re-querying the API. Off by default.

Value

a data frame of information

Examples

```
## Not run:
researchdata4tu_info("https://doi.org/10.4121/16766929.v1")

## End(Not run)
```

researchdata4tu_links	<i>Find 4TU.ResearchData Links in Papers</i>
-----------------------	--

Description

Get all 4TU.ResearchData links: real hyperlinks from the paper’s own url table, plus a body-text fallback for a BARE mention (a DOI like "10.4121/16766929" is routinely cited without any URL scheme at all) that the source PDF/HTML never encoded as an actual hyperlink – the url table only ever contains links the source document itself made clickable. Same two-tier approach `zenodo_links()` uses.

Usage

```
researchdata4tu_links(paper)
```

Arguments

paper	a paper object or paperlist object
-------	------------------------------------

Value

a table with the 4TU.ResearchData url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
researchdata4tu_links(psychsci)

## End(Not run)
```

researchdata4tu_pat	<i>Set or get a 4TU.ResearchData API token</i>
---------------------	--

Description

4TU.ResearchData issues its own personal access tokens (from account settings on data.4tu.nl) – a Figshare.com token is meaningless here, so this is stored separately from [figshare_pat\(\)](#). A token is optional for reading public articles (used here only to raise rate limits or read private ones).

Usage

```
researchdata4tu_pat(pat = NULL)
```

Arguments

pat	the token to set, or NULL to get the current token
-----	--

Details

Store it as the RESEARCHDATA4TU_PAT environment variable so it is read every time R starts.

Value

the current token (character; "" when none is set)

Examples

```
researchdata4tu_pat() # returns "" unless a token is set
```

reshare_file_download *Download all files from a ReShare deposit*

Description

Creates a directory for the deposit and downloads all of its files. Returns (invisibly) a data frame with file info.

Usage

```
reshare_file_download(
  reshare_id,
  download_to = ".",
  max_file_size = 10,
  max_download_size = 100,
  unzip_types = NULL,
  pb = NULL
)
```

Arguments

reshare_id	a ReShare eprint ID, URL, or DOI
download_to	path to download to
max_file_size	maximum file size to download (in MB) - set to NULL or Inf for no restrictions
max_download_size	maximum total size to download - set to NULL or Inf for no restrictions
unzip_types	file categories to extract from .zip files in the deposit rather than downloading the zip whole, named as data_classify_files() names them: "data", "code", "materials", "documentation", "output", "unknown". More than one may be given. NULL (the default) downloads zips whole and leaves them zipped, as before. max_file_size applies to each extracted member rather than to the archive.
pb	a progress bar passed from another function

Details

Like Figshare, ReShare's EPrints REST API documents no whole-deposit bulk-download endpoint – each file is served from its own `.../id/file/<fileid>` URL, so this always downloads file by file. A .zip already present in the deposit (ReShare's own documents are frequently zips of related files, e.g. all interviews for one country) is still handled the same way [dataverse_file_download\(\)](#) handles one: fetched whole and left as a zip unless unzip_types asks for only specific members out of it.

You can limit downloads to only files under a specific size (defaults to 10MB) and only a maximum download size (largest files will be omitted until total size is under the limit). Omitted files will be listed as messages in verbose mode, and included in the returned data frame with the downloaded column value set to FALSE.

Value

data frame of file info. When members are extracted from a zip, its row reports the zip with extracted giving the number of members written; size_on_disk and checksum_ok are then NA, because what is on disk is the members rather than the archive ReShare published a size and MD5 for.

Examples

```
## Not run:
  reshare_file_download("854001")

# take only the data files out of any zip in the deposit
  reshare_file_download("854001", unzip_types = "data")

## End(Not run)
```

reshare_info

*Retrieve info from ReShare by URL***Description**

Retrieve info from ReShare by URL

Usage

```
reshare_info(reshare_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

reshare_url	a ReShare URL or DOI, or a table containing them (e.g., as created by reshare_links())
id_col	the index or name of the column that contains ReShare URLs, if reshare_url is a table
pb	a progress bar passed from another function
cache	if TRUE, reuse a previously cached listing for a deposit already looked up (see repo_info_cache()) instead of re-querying the API. Off by default.

Value

a data frame of information

Examples

```
## Not run:
  reshare_info("https://doi.org/10.5255/UKDA-SN-854001")

## End(Not run)
```

reshare_links

Find ReShare Links in Papers

Description

Get all ReShare links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE mention (a DOI like "10.5255/UKDA-SN-854001" is routinely cited without any URL scheme at all) that the source PDF/HTML never encoded as an actual hyperlink – the url table only ever contains links the source document itself made clickable. Same two-tier approach zenodo_links() uses.

Usage

```
reshare_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the ReShare url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
reshare_links(psychsci)

## End(Not run)
```

retractionwatch

RetractionWatch data

Description

DOIs and nature of statements from the RetractionWatch database. Use rw_date() to find the date it was downloaded, and rw_update() to update it.

Usage

```
retractionwatch()
```

```
rw()
```

Format

A data frame with 44784+ rows and 2 columns:

doi Document ID

retractionwatch Nature of note(s)

Value

a data frame

Source

<https://api.labs.crossref.org/data/retractionwatch>

Examples

```
retractionwatch() |> head()
```

rw_date

Get date retractionwatch was updated

Description

Get date retractionwatch was updated

Usage

```
rw_date()
```

Value

the date

Examples

```
rw_date()
```

rw_update	<i>Update retractionwatch</i>
-----------	-------------------------------

Description

metacheck comes with a built-in data frame called `retractionwatch`. We update it regularly, but you can use this function to download the newest version. The download is >50MB, but this function will summarise the information into a smaller version (~0.5 MB) and delete the original file.

Usage

```
rw_update()
```

Value

the path to the data frame (invisibly)

scroll_table	<i>Make Scroll Table</i>
--------------	--------------------------

Description

A helper function for making module reports.

Usage

```
scroll_table(  
  table,  
  colwidths = "auto",  
  maxrows = 2,  
  escape = FALSE,  
  column = "body"  
)
```

Arguments

- | | |
|-----------|--|
| table | the data frame to show in a table, or a vector for a list |
| colwidths | set column widths as a vector of px (number > 1) or percent (numbers <= 1) |
| maxrows | if the table has more rows than this, paginate |
| escape | whether or not to escape the DT (necessary if using raw html) |
| column | which quarto column to show tables in |

Details

See [quarto article layout](#) for column options. The most common are "body" (centre column), "page" (span all columns), and "margin" (only in right margin).

To set colwidths, use a numeric or character vector. For a numeric vector, numbers greater than 1 will be interpreted as pixels, less than 1 as percents. Character vectors will be passed as is (e.g., "3em"). If you only want to specify some columns, set the others to NA, like `c(200, NA, 200, NA)`.

Value

the markdown R chunk to create this table

Examples

```
scroll_table(LETTERS)
```

stat_output_json	<i>Serialise extracted result tables as a structured statistical-output document</i>
------------------	--

Description

Builds a complete statistical-output document from the result tables of ONE `.jasp/.omv` file or executed R script: one document per source file, with an `analyses` array (one element per analysis heading/table) each carrying a `results` array (one element per result row) of STATO-typed statistic values (via `stato_type_column()`). This is a metacheck-native schema — a flat, self-describing document in the character of OSD/DDI-Lifecycle/Psych-DS — not a repurposing of an unrelated external standard’s vocabulary; there is no external schema this validates against.

Usage

```
stat_output_json(tables, paper_id = "metacheck", source_file = NA_character_)
```

Arguments

tables	the list returned by <code>read_stat_tables()</code> or <code>read_r_output()</code>
paper_id	paper id / DOI, recorded on the document
source_file	basename of the originating <code>.jasp/.omv</code> , or the R script name for run-R output, recorded as <code>source_file</code> and used to derive <code>source_format</code> and the <code>result_id</code> prefix (see below)

Value

a list ready to serialise with `jsonlite::toJSON(auto_unbox = TRUE)`, with elements `schema`, `schema_version`, `paper_id`, `source_file`, `source_format`, and `analyses` (a list of `list(analysis, results)`, where each result is `list(result_id, row_label, values)` and `values` is a named list keyed by the statistic's own short name, e.g. `t/df/p/d`, each a `list(value, stato_label, stato_iri)` — `stato_label/stato_iri` omitted when `stato_type_column()` found no class for that statistic). `NULL` when there are no tables or no table yields any typed value. `result_id` identifies the CODE ROW that produced it — see `stat_results_long()` for the id format (shared between both serialisations of one extraction).

<code>stat_output_validate</code>	<i>Validate a statistical-output document's native shape</i>
-----------------------------------	--

Description

Checks that a `stat_output_json()` document has the required top-level fields, that `analyses` is a list of `list(analysis, results)`, and that every result carries a `result_id` and a non-empty `values` object whose entries each have a value. This is a native structural check, not an executed external JSON Schema — there is no external standard this document conforms to, by design (see the file header comment). Mirrors `behavarse_validate()`'s and `psychds-validate.R`'s hand-rolled approach.

Usage

```
stat_output_validate(doc)
```

Arguments

<code>doc</code>	a statistical-output document as an R list (as from <code>stat_output_json()</code>), or a length-1 character path / JSON string to parse first.
------------------	---

Value

a list with `valid` (TRUE when no issues), `issues` (a character vector of problem descriptions), and `summary` (`n_errors`, `n_analyses`, `n_results`).

stat_output_write	<i>Write extracted statistical output to a dedicated folder</i>
-------------------	---

Description

Writes reproducibility_check's accumulated stat_output (one element per source file: JASP/jamovi files via `read_stat_tables()`, executed R scripts via `read_r_output()`) to <root>/statistical_output/, a folder sibling to the materialised data/ — so the extracted statistics sit alongside the data and code that produced them. Two views are written from the SAME extraction: one combined results_long.csv (every source file's `stat_results_long()` rows stacked, one row per extracted statistic, easiest to load and filter), and one <codename>.statistical_output.json per source file (its `stat_output_json()` document). Called from inside reproducibility_check itself (not from the later psychds/scienceverse conversion step), so the folder exists at the same point in the pipeline the data/code files do.

Usage

```
stat_output_write(stat_output, root)
```

Arguments

stat_output	a list as accumulated by reproducibility_check — each element a list with file (source file name), json (<code>stat_output_json()</code> 's document), and long (<code>stat_results_long()</code> 's table)
root	the materialised layout root (the same directory data/ is copied into); statistical_output/ is created under it

Value

the path to statistical_output/, invisibly, or NULL (nothing written) when stat_output is empty or carries no rows/documents.

stat_results_long	<i>Flatten extracted result tables into one queryable long data frame</i>
-------------------	---

Description

Turns the nested output of `read_stat_tables()` into a tidy long table: one row per (analysis, table, result-row, statistic) cell, carrying the STATO typing. This is the form stored in the scienceverse SQLite database, and the basis for later matching of extracted results against the manuscript.

Usage

```
stat_results_long(
  tables,
  paper_id = NA_character_,
  source_file = NA_character_
)
```

Arguments

tables	the list returned by <code>read_stat_tables()</code> or <code>read_r_output()</code>
paper_id	optional paper id / DOI to stamp on every row
source_file	optional name of the file the tables were extracted from (e.g. the <code>.jasp/.omv</code> basename, or the script name for run-R output), recorded on every row as provenance for downstream matching, and as the <code>result_id</code> prefix (see below)

Value

a data.frame with columns `paper_id`, `source_file`, `result_id`, `analysis`, `table_title`, `row_label`, `statistic`, `stato_label`, `stato_iri`, and `value`. `result_id` identifies ONE CELL — one statistic of one code row — sanitised to lower-case letters/digits/underscores: `<source_file>_l<line>_<line_seq>_r<row>_<statistic>` for an executed R script (from `read_r_output()`'s echo-based line attribution), or `<source_file>_t<table_index>_r<row>_<statistic>` for a JASP/jamovi extraction (no source line exists there, so the table's ordinal position in the rendered document stands in for one); `row` is the 1-based position of this result within its table, and `statistic` is that cell's own column header, sanitised the same way and disambiguated with a trailing counter on the rare repeat within one row (a coefficients matrix with a duplicated header). Every ROW's shared prefix (everything before the trailing `_<statistic>`) is `.stat_test_id()`'s locator, so cells from the same row share that prefix — only the final segment tells them apart. Empty frame (same columns) when there is nothing to flatten.

stato_type_column	<i>Type a result-table column by its header, with a guaranteed fallback</i>
-------------------	---

Description

Maps a JASP/jamovi/R result-table column header to a STATO ontology class when one exists, then to a metacheck-minted statistic term for the standard statistics STATO has no class for, and otherwise falls back to the header text itself as an untyped label. Every column therefore gets a type, so no column is ever dropped from the statistical-output export.

Usage

`stato_type_column(header, call_fn = NULL)`

Arguments

header	the column header string as rendered in the result table
call_fn	optional name of the R function that produced this result (e.g. <code>"shapiro.test"</code>), as carried by <code>read_r_output()</code> . When supplied it disambiguates headers that name different quantities in different tests – R prints a bare <code>W</code> for both Shapiro-Wilk's <code>W</code> and the Wilcoxon rank sum – which no header-only lookup can resolve. Ignored for GUI formats, which have no call.

Details

termSource says which vocabulary answered: "STATO" for a verified STATO class, "metacheck" for a minted term (see .MC_STAT_LABELS for what each one means and why STATO has no equivalent), or "" when the header could not be typed at all.

Value

a list with annotationValue (the canonical label, or the header text when unmapped), termSource ("STATO", "metacheck", or ""), and termAccession (the term IRI, or "").

stats	<i>Check Stats</i>
-------	--------------------

Description

Check Stats

Usage

```
stats(text, ...)
```

Arguments

text	the search table (or list of paper objects)
...	arguments to pass to statcheck()

Value

a table of statistics

Examples

```
paper <- demopaper()
stats(paper)
```

test_paper	<i>Test paper</i>
------------	-------------------

Description

Create a paper object with the specified text (mainly for testing/demos).

Usage

```
test_paper(text = LETTERS, url = character(0))
```

Arguments

text	a vector of text to add
url	a vector of URLs to add

Value

a paper object

Examples

```
# to test a paper with a specific URL
p <- test_paper("https://osf.io/abcde")
```

text_expand	<i>Expand text</i>
-------------	--------------------

Description

If you have a table resulting from `text_search()` or a module return object, you can expand the text column to the full sentence, paragraph, or section. You can also set `plus` and `minus` to append and prepend sentences to the result (only when `expand_to` is "sentence").

Usage

```
text_expand(
  results_table,
  paper,
  expand_to = c("sentence", "paragraph", "div", "section"),
  plus = 0,
  minus = 0
)

expand_text(
  results_table,
```

```
paper,
expand_to = c("sentence", "paragraph", "div", "section"),
plus = 0,
minus = 0
)
```

Arguments

- results_table the table to expand
- paper a metacheck paper object or a list of paper objects to look up the expanded text from
- expand_to whether to expand to the sentence, paragraph, div, or section level
- plus append additional sentences after the target expansion
- minus prepend additional sentences before the target expansion

Value

a results table with the expanded text

Examples

```
# single paper search
paper <- demopaper()
res_tbl <- text_search(paper, "p =", return = "match")
expanded <- text_expand(res_tbl, paper)

# multiple paper search
## Not run:
papers <- papers_load("psychsci", cache = TRUE)
res_tbl <- text_search(papers, "replicate")
expanded <- text_expand(res_tbl, papers, plus = 1, minus = 1)

## End(Not run)
```

text_search	<i>Search text</i>
-------------	--------------------

Description

Search the text of a paper or list of paper objects. Also works on the table results of a text_search() call.

Usage

```

text_search(
  paper,
  pattern = ".*",
  return = c("sentence", "paragraph", "section", "header", "match", "paper_id"),
  ignore.case = TRUE,
  fixed = FALSE,
  perl = FALSE,
  exclude = FALSE,
  search_header = FALSE,
  include_refs = FALSE
)

search_text(
  paper,
  pattern = ".*",
  return = c("sentence", "paragraph", "section", "header", "match", "paper_id"),
  ignore.case = TRUE,
  fixed = FALSE,
  perl = FALSE,
  exclude = FALSE,
  search_header = FALSE,
  include_refs = FALSE
)

```

Arguments

<code>paper</code>	a paper object or a list of paper objects
<code>pattern</code>	the regex pattern to search for, if a vector with length > 1, the patterns will be searched separately and combined
<code>return</code>	the kind of text to return, the full sentence, paragraph, header, or section that the text is in, or just the (regex) match, or all body text for a paper (<code>paper_id</code>)
<code>ignore.case</code>	whether to ignore case when text searching
<code>fixed</code>	logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
<code>perl</code>	logical. Should Perl-compatible regexps be used?
<code>exclude</code>	should matches be included or excluded
<code>search_header</code>	also search the header
<code>include_refs</code>	whether to include the reference section in the search

Details

The `section` argument can take a vector of section names, or a PERL regular expression (use `".*"` to match all sections). Possible section types are abstract, intro, method, results, discussion, references, acknowledgment, funding, endnote, footnote, table, figure, and unknown. The default includes all sections except references, tables and figures.

Value

a data frame of matches

Examples

```
paper <- demopaper()
all_text <- text_search(paper)
study <- text_search(paper, "study")
equations <- text_search(paper, "\\b\\S+\\s*(=|<)\\s*[0-9\\.]+", return = "match")
no_numbers <- text_search(paper, "\\d", exclude = TRUE)
```

validate	<i>Validate a module's output against coded ground truth</i>
----------	--

Description

Runs module on a set of texts you have already coded by hand, then checks its output against your coding. Each text in gt becomes its own test paper; the module's output table (compare) is joined back to gt by paper_id and text, and for every column that appears in both, a <column>.valid logical column is added marking where the module's value matches your ground truth. Pass the result to [accuracy\(\)](#) to summarise agreement (e.g. sensitivity, specificity, d-prime).

Usage

```
validate(gt, module, compare = "table")
```

Arguments

- gt a vector of texts to code, or a data frame with paper_id and text columns plus one column per ground-truth value to compare against the module's output (column names must match the module output table)
- module the name of the module to run and validate
- compare name of the list item in the module's output to compare against gt (usually "table", the module's full row-level results)

Value

gt full-joined with the module's compare table (columns suffixed .gt and .mod where names collide), plus one <column>.valid logical column per compared column

Examples

```
validate("p < .05", "stat_p_exact")
```

zenodo_file_download *Download all Zenodo Project Files*

Description

Creates a directory for the Zenodo ID and downloads all of the files using a folder structure from the Zenodo project nodes and file storage structure. Returns (invisibly) a data frame with file info.

Usage

```
zenodo_file_download(
  zenodo_id,
  download_to = ".",
  max_file_size = 10,
  max_download_size = 100,
  unzip_types = NULL,
  pb = NULL
)
```

Arguments

zenodo_id	an Zenodo ID or URL
download_to	path to download to
max_file_size	maximum file size to download (in MB) - set to NULL or Inf for no restrictions
max_download_size	maximum total size to download - set to NULL or Inf for no restrictions
unzip_types	file categories to extract from .zip files in the record rather than downloading the zip whole, named as data_classify_files() names them: "data", "code", "materials", "documentation", "output", "unknown". More than one may be given. NULL (the default) downloads zips whole and leaves them zipped, as before. max_file_size applies to each extracted member rather than to the archive.
pb	a progress bar passed from another function

Details

You can limit downloads to only files under a specific size (defaults to 10MB) and only a maximum download size (largest files will be omitted until total size is under the limit). Omitted files will be listed as messages in verbose mode, and included in the returned data frame with the downloaded column value set to FALSE.

A .zip in the record is normally fetched whole and left as a zip. Set unzip_types to pull only the files you want out of it instead: Zenodo serves per-file URLs with byte-range support and each member of a zip is compressed separately, so the wanted members can be read out of the archive while the rest is never transferred. A zip of stimuli and three data files then costs the three data files. When the archive cannot be read that way (the listing fails, or the host refuses ranges) it is downloaded whole as before, so this never loses a file – it only avoids transferring one.

Value

data frame of file info. When members are extracted from a zip, its row reports the zip with extracted giving the number of members written; size_on_disk and checksum_ok are then NA, because what is on disk is the members rather than the archive Zenodo published a size and MD5 for.

Examples

```
## Not run:
  zenodo_file_download("2591593")

# take only the data files out of any zip in the record
  zenodo_file_download("2591593", unzip_types = "data")

## End(Not run)
```

zenodo_info	<i>Retrieve info from Zenodo by URL</i>
-------------	---

Description

Retrieve info from Zenodo by URL

Usage

```
zenodo_info(zenodo_url, id_col = 1, pb = NULL, cache = FALSE)
```

Arguments

zenodo_url	an Zenodo URL, or a table containing them (e.g., as created by zenodo_links())
id_col	the index or name of the column that contains Zenodo URLs, if id is a table
pb	a progress bar passed from another function
cache	if TRUE, reuse a previously cached listing for a record already looked up (see repo_info_cache() instead of re-querying Zenodo's API. Off by default.

Value

a data frame of information

Examples

```
## Not run:
# get info on one zenodo link
  zenodo_info("https://doi.org/10.5281/zenodo.18648142")

## End(Not run)
```

zenodo_links

*Find Zenodo Links in Papers***Description**

Get all Zenodo links: real hyperlinks from the paper's own url table, plus a body-text fallback for a BARE mention (a DOI like "10.5281/zenodo.1234567" is routinely cited without any URL scheme at all) that the source PDF/HTML never encoded as an actual hyperlink — the url table only ever contains links the source document itself made clickable. Same two-tier approach `github_links()` uses for GitHub.

Usage

```
zenodo_links(paper)
```

Arguments

paper a paper object or paperlist object

Value

a table with the Zenodo url in the first (text) column

Examples

```
## Not run:
psychsci <- papers_load("psychsci", cache = TRUE)
zenodo_links(psychsci)

## End(Not run)
```

zenodo_pat

*Set or get the Zenodo personal access token***Description**

Use `zenodo_pat()` to get the token used to authorise Zenodo uploads, or `zenodo_pat("your-token")` to set it for the rest of the session. The sandbox and the real Zenodo are separate services with separate accounts, so they need separate tokens and are stored separately here.

Usage

```
zenodo_pat(pat = NULL, sandbox = TRUE)
```

Arguments

pat	the token to set, or NULL to get the current token
sandbox	whether the token is for the sandbox server

Details

To create a token, sign in and go to Applications, then "Personal access tokens", then "New token". Give it a name and tick the `deposit:write` scope (to create depositions and add files) and `deposit:actions` (to publish). Copy the token immediately, as Zenodo does not show it again.

- sandbox token: <https://sandbox.zenodo.org/account/settings/applications/tokens/new/>
- real token: <https://zenodo.org/account/settings/applications/tokens/new/>

The sandbox needs its own account, registered at <https://sandbox.zenodo.org>; a real zenodo.org login does not work there.

Store them as lines in your `.Renviro`n file (open it with `usethis::edit_r_environ()`) so they are read every time R starts:

```
ZENODO_SANDBOX_PAT="replace-with-your-sandbox-token"
ZENODO_PAT="replace-with-your-real-token"
```

Value

the current token (character; "" when none is set)

Examples

```
zenodo_pat() # returns "" unless a token is set
```

zenodo_upload	<i>Upload Folders to Zenodo</i>
---------------	---------------------------------

Description

Creates a Zenodo deposition for each folder, uploads every file in it, and attaches metadata. Designed to take the output of `osf_file_download()` directly, so a whole OSF account can be archived on Zenodo in two steps:

Usage

```
zenodo_upload(
  folders,
  sandbox = TRUE,
  zenodo_pat = NULL,
  publish = FALSE,
  license = "cc-by-4.0",
```

```

upload_type = "dataset",
metadata = NULL,
as_zip = TRUE,
split_materials = "materials",
upload_osf_metadata = TRUE,
max_file_size = NULL,
ask = TRUE,
pb = NULL
)

```

Arguments

folders	a data frame returned by <code>osf_file_download()</code> , or a character vector of paths to folders to upload (one deposition per folder)
sandbox	whether to upload to the Zenodo sandbox (the default) rather than the real zenodo.org
zenodo_pat	a Zenodo personal access token. Defaults to whatever <code>zenodo_pat()</code> returns for this server. See <code>zenodo_pat()</code> for how to create one.
publish	whether to publish each deposition. FALSE (the default) leaves it as a draft for you to check and publish by hand.
license	the Zenodo license identifier to use when the OSF project has no license Zenodo recognises
upload_type	the Zenodo resource type (e.g. "dataset", "publication", "software")
metadata	a list of extra metadata fields to add to every deposition, overriding what was taken from the OSF
as_zip	whether to upload each folder as a single ZIP archive instead of as individual files. Zenodo has no folders. Its documentation says so directly – "Zenodo does not support uploading and organising files into folders/directories. Instead, you can create a ZIP archive and upload it, in which case Zenodo will display the file structure inside the ZIP" – and the API agrees: a file name containing "/" is rejected, and a file record has no path field. So there is a choice. TRUE (the default) keeps the folder structure exactly, which is what an archive is for: a reader gets the project as the author arranged it, rather than a heap of files they must sort out. The usual objection – that reaching one small file means downloading everything – is what <code>split_materials</code> answers, by putting the bulky part in a separate archive. FALSE uploads the files separately, which keeps each one visible, previewable and downloadable on its own; the folder structure is lost, though file names are kept short and only prefixed where two files would otherwise collide. That suits a small flat folder, where the structure carries no meaning worth preserving.
split_materials	which CATEGORIES of file to put in their own archive when <code>as_zip = TRUE</code> . Defaults to "materials", which is usually the bulk of a repository's size and the part a reader is least likely to need.

Categories are the same six `repo_check()` reports – data, code, materials, documentation, output, unknown – decided by the same classifier, so a file lands in the same category whether it is being checked or archived. Pass more than one to separate them further, for example `c("materials", "output")`.

Every archive stores the same project folder at its root, so unzipping all of them rebuilds the original tree exactly, while unzipping only the main one gives everything except the split categories, each file still in its right place. That lets somebody take the data and code without a multi-gigabyte stimulus set.

Set NULL or FALSE for a single archive holding everything. Ignored unless `as_zip = TRUE`.

`upload_osf_metadata`

whether to upload the `_osf_metadata` folder that `osf_file_download()` saves inside each project – the wiki pages, the activity log, and the project record. TRUE by default, so the archive on Zenodo describes itself. Set FALSE to deposit only the data files; the metadata is still read either way, since that is where the deposition's title, licence and authors come from.

`max_file_size` largest file to upload, in MB; larger files are skipped and reported. Set to NULL for no limit.

`ask` whether to show a summary and ask for confirmation before uploading anything. Ignored in a non-interactive session, where the upload proceeds without asking.

`pb` a progress bar passed from another function

Details

```
osf_file_download("4i578", download_to = "my_osf") |>
  zenodo_upload()
```

Value

a data frame with one row per folder: the folder path, the Zenodo deposition id, its DOI, the URL to review it, how many files were uploaded and skipped, and whether it was published

Safety

Uploads go to the Zenodo **sandbox** by default. The sandbox is a complete separate copy of Zenodo, run for testing at <https://sandbox.zenodo.org>. It has its own accounts and its own tokens, and the DOIs it mints use a test prefix that never resolves publicly, so nothing you do there is permanent. Set `sandbox = FALSE` only once a sandbox run looks right.

Depositions are left as unpublished **drafts** (`publish = FALSE`). A draft can be edited or deleted freely. Review each draft in your Zenodo account, then press Publish there when you are satisfied. Publishing on the real `zenodo.org` is irreversible: the record and its DOI are permanent and cannot be deleted.

Examples

```
## Not run:
# download one OSF project and put it on the sandbox as a draft
osf_file_download("6nt4v", download_to = "archive") |>
```

```
zenodo_upload()

# upload a folder you already have
zenodo_upload("archive/6nt4v")

# once the sandbox run looks right, upload for real (still a draft)
zenodo_upload("archive/6nt4v", sandbox = FALSE)

## End(Not run)
```

zip_peek_cache_clear *Clear the on-disk zip-peek cache*

Description

Deletes all cached zip listings (see [zip_peek\(\)](#)'s cache argument). Safe at any time – a cleared entry is simply re-peeked over HTTP the next time it is needed.

Usage

```
zip_peek_cache_clear()
```

Value

the number of cache entries removed, invisibly

Index

* datasets

- emojis, [46](#)
- .mplus_export_syntax(), [69](#)
- .osf_node_data(), [108](#)
- .report_table_simple_max_rows, [135](#)
- .repro_bioc_packages(), [141](#)
- .repro_classify_install_message(), [142](#), [143](#)
- .repro_docker_default_image_py, [144](#), [154](#)
- .repro_docker_image_for(), [142](#), [151](#)
- .repro_format_call_refs(), [146](#), [147](#)
- .repro_missing_fn_pkgs(), [155](#)
- .repro_normalize_basename(), [149](#)
- .repro_package_plan(), [145](#), [155](#)
- .repro_redirect_writes(), [155](#)
- .repro_simple_string_vars(), [147](#)
- .smcl_render(), [71](#)
- .spv_decode_chart(), [48](#)
- .spv_decode_legacy_data(), [70](#)
- .spv_decode_legacy_table(), [70](#)
- .spv_decode_light_table(), [70](#)
- .spv_export_syntax(), [69](#)
- accuracy, [6](#)
- accuracy(), [173](#)
- add_bib_match, [7](#)
- aspredicted_info, [8](#)
- aspredicted_links, [8](#)
- callr::r(), [149](#)
- causal_relations, [9](#)
- code_abs_path, [10](#)
- code_extract_py, [11](#)
- code_extract_py(), [12](#)
- code_extract_qmd_py, [12](#)
- code_extract_r, [13](#)
- code_extract_r(), [11](#), [12](#)
- code_file_refs, [13](#)
- code_file_refs(), [11](#), [140](#), [148](#)
- code_install_packages, [14](#)
- code_lang, [15](#)
- code_lang(), [11](#), [12](#)
- code_library_lines, [16](#)
- code_library_lines(), [16](#)
- code_library_names, [16](#)
- code_library_names(), [11](#), [138](#), [139](#)
- code_line_stats, [18](#)
- code_packages, [18](#)
- code_parse_r, [19](#)
- code_read, [20](#)
- code_remove_comments, [20](#)
- code_remove_comments(), [11](#), [14](#), [21](#)
- code_setwd, [21](#)
- code_setwd(), [14](#)
- collapse_section, [22](#)
- convert, [23](#)
- convert_bibr, [23](#), [24](#)
- convert_grobid, [25](#)
- crossref_doi, [27](#)
- crossref_query, [28](#)
- data_classify_files(), [32](#), [40](#), [53](#), [157](#), [160](#), [174](#)
- datacite_doi, [29](#)
- dataone_info, [30](#)
- dataone_links, [30](#)
- dataone_links(), [30](#)
- dataverse_file_download, [31](#)
- dataverse_file_download(), [40](#), [160](#)
- dataverse_info, [33](#)
- dataverse_links, [33](#)
- dataverse_links(), [33](#)
- dataverse_pat, [34](#)
- demofile, [35](#)
- demopaper, [35](#)
- doi_clean, [36](#)
- doi_lookup, [36](#)
- doi_resolves, [37](#)
- doi_valid_format, [38](#)

- download_repo_files, [44](#)
- download_repo_files(), [116](#), [145](#)
- dryad_auth, [38](#)
- dryad_auth(), [43](#)
- dryad_file_download, [39](#)
- dryad_file_download(), [42](#)
- dryad_info, [41](#)
- dryad_info(), [40](#), [129](#)
- dryad_links, [42](#)
- dryad_links(), [41](#)
- dryad_pat, [42](#)
- dryad_pat(), [38–40](#)
- dspace7_file_download, [44](#)
- dspace7_links, [44](#)
- dspace_links, [43](#)
- email, [45](#)
- emojis, [46](#)
- expand_text(text_expand), [170](#)
- export_jasp_html, [46](#)
- export_jasp_html(), [48](#)
- export_mplus_html, [47](#)
- export_omv_html, [47](#)
- export_omv_html(), [48](#)
- export_spv_html, [48](#)
- export_spv_html(), [48](#)
- export_stata_smcl_html, [48](#)
- export_stata_smcl_html(), [47](#)
- extract_eq, [49](#)
- extract_eq(), [51](#)
- extract_p_values, [50](#)
- extract_tests, [50](#)
- extract_urls, [51](#)
- fig_image_view, [52](#)
- figshare_file_download, [52](#)
- figshare_file_download(), [40](#), [156](#)
- figshare_info, [54](#)
- figshare_info(), [157](#)
- figshare_links, [55](#)
- figshare_links(), [54](#)
- figshare_pat, [55](#)
- figshare_pat(), [157](#), [159](#)
- file_category, [56](#)
- filetype, [57](#)
- FLoRA, [57](#)
- FLoRA_date, [58](#)
- FLoRA_update, [58](#)
- format_bib_authors, [59](#)
- format_ref, [59](#)
- fsd_info, [60](#)
- fsd_links, [61](#)
- get_prev_outputs, [61](#)
- github_files, [62](#)
- github_info, [63](#)
- github_languages, [63](#)
- github_links, [64](#)
- github_readme, [64](#)
- github_repo, [65](#)
- gitlab_links, [66](#)
- gitlab_pat, [66](#)
- gitlab_repo, [67](#)
- grobid_to_bibr, [68](#)
- import_jasp, [68](#)
- import_jasp(), [70](#)
- import_mplus_output, [69](#)
- import_mplus_output(), [47](#)
- import_omv, [70](#)
- import_spv, [70](#)
- import_spv(), [48](#)
- import_stata_smcl, [71](#)
- import_stata_smcl(), [48](#)
- json_expand, [72](#)
- lastlog, [73](#)
- ldbase_info, [73](#)
- ldbase_links, [74](#)
- ldbase_links(), [74](#)
- link, [75](#)
- llm, [75](#)
- llm(), [77](#)
- llm_cache, [77](#)
- llm_cache(), [77](#)
- llm_cache_clear, [77](#)
- llm_cache_clear(), [87](#)
- llm_max_calls, [78](#)
- llm_max_tokens, [78](#)
- llm_model, [79](#)
- llm_model_list, [79](#)
- llm_reasoning, [80](#)
- llm_timeout, [81](#)
- llm_use, [82](#)
- llm_use(), [134](#)
- local_files, [82](#)
- logger, [83](#)

- logger(), 73
- manifest_merge, 83
- match_reported_output, 84
- mendeley_info, 85
- mendeley_links, 86
- mendeley_links(), 86
- metacheck_cache_info, 87
- module_help, 88
- module_help(), 89
- module_info, 88
- module_install, 89
- module_install(), 90, 91, 93
- module_installed, 90
- module_installed(), 89, 93
- module_list, 90
- module_list(), 89, 90
- module_remove, 91
- module_remove(), 89
- module_report, 91
- module_report(), 89
- module_run, 92
- module_run(), 89
- module_template, 93
- module_template(), 89
- module_update, 93
- module_update(), 89
- openalex_doi, 94
- openalex_query, 96
- osf_api_check, 97
- osf_app, 98
- osf_cache_clear, 98
- osf_cache_clear(), 104
- osf_check_id, 99
- osf_delay, 99
- osf_file_download, 100
- osf_file_download(), 98, 108, 134, 177–179
- osf_get_all_pages, 103
- osf_info, 104
- osf_links, 105
- osf_pat, 105
- osf_pat(), 98, 101
- osf_preprint_list, 106
- osf_type, 107
- osf_user_projects, 108
- osf_user_projects(), 98, 100, 102
- paper_id, 109
- paper_table, 109
- paper_validate, 110
- paper_write, 111
- paper_write(), 68
- papers_available, 111
- papers_available(), 112–114
- papers_load, 112
- papers_load(), 111
- papers_metadata, 113
- papers_remove, 114
- papers_remove(), 112
- path_sanitize, 114
- pb(), 74
- plural, 115
- psycharchives_file_download, 116
- psycharchives_info, 116
- psycharchives_links, 117
- pubpeer_comments, 118
- rbox_file_download, 118
- rbox_info, 119
- rbox_links, 119
- read, 120
- read_r_output, 121
- read_r_output(), 71, 150, 165, 167, 168
- read_stat_tables, 122
- read_stat_tables(), 11, 69–71, 121, 165, 167, 168
- ref_table, 123
- regcheck_compare, 123
- regcheck_setup_local, 125
- regcheck_setup_local(), 126
- regcheck_start_local, 126
- regcheck_start_local(), 125, 127
- regcheck_stop_local, 127
- repo_cache_clear, 127
- repo_cache_clear(), 87, 128, 129
- repo_cache_dir, 128
- repo_cache_dir(), 87, 127, 129
- repo_cache_size, 129
- repo_cache_size(), 127, 128
- repo_info_cache, 129
- repo_info_cache(), 30, 33, 41, 54, 60, 86, 87, 104, 116, 130, 158, 161, 175
- repo_info_cache_clear, 130
- repo_info_cache_clear(), 87, 130
- report, 130
- report(), 22, 89, 134, 136

report_app, 132
 report_module_run, 132
 report_qmd, 133
 report_repository, 134
 report_repository(), 136
 report_table, 135
 report_table(), 136
 report_type, 136
 report_type(), 22, 131, 134, 135
 repro_defined_vars, 137
 repro_dependencies, 138
 repro_dependencies(), 139, 141, 142
 repro_dependencies_py, 139
 repro_dependencies_py(), 143, 144
 repro_docker_available, 140
 repro_file_io, 140
 repro_file_io(), 148, 155
 repro_install_deps, 141
 repro_install_deps(), 142, 143, 150
 repro_install_deps_docker, 142
 repro_install_deps_docker(), 151
 repro_install_deps_py, 143
 repro_install_deps_py(), 144, 152
 repro_install_deps_py_docker, 144
 repro_install_deps_py_docker(), 154
 repro_materialize_layout, 144
 repro_materialize_layout(), 147, 151
 repro_missing_inputs, 145
 repro_rewrite_paths, 146
 repro_run_order, 148
 repro_run_order(), 140
 repro_run_scripts, 149
 repro_run_scripts(), 150–153
 repro_run_scripts_docker, 150
 repro_run_scripts_docker(), 142
 repro_run_scripts_py, 152
 repro_run_scripts_py(), 153, 154
 repro_run_scripts_py_docker, 153
 repro_run_scripts_py_docker(), 144, 153
 repro_write_scripts, 154
 repro_write_scripts(), 147
 researchdata4tu_file_download, 156
 researchdata4tu_file_download(), 53
 researchdata4tu_info, 157
 researchdata4tu_info(), 54
 researchdata4tu_links, 158
 researchdata4tu_links(), 158
 researchdata4tu_pat, 159
 researchdata4tu_pat(), 157
 reshare_file_download, 160
 reshare_info, 161
 reshare_links, 162
 reshare_links(), 161
 retractionwatch, 162
 rw(retractionwatch), 162
 rw_date, 163
 rw_update, 164

 scroll_table, 164
 scroll_table(), 137
 search_text(text_search), 171
 stat_output_json, 165
 stat_output_json(), 122, 166, 167
 stat_output_validate, 166
 stat_output_write, 167
 stat_results_long, 167
 stat_results_long(), 84, 122, 166, 167
 stato_type_column, 168
 stato_type_column(), 121, 165, 166
 stats, 169
 Sys.time(), 150–152, 154

 test_paper, 170
 text_expand, 170
 text_search, 171

 validate, 173

 zenodo_file_download, 174
 zenodo_file_download(), 32, 40
 zenodo_info, 175
 zenodo_info(), 129
 zenodo_links, 176
 zenodo_pat, 176
 zenodo_pat(), 178
 zenodo_upload, 177
 zenodo_upload(), 102
 zip_peek(), 180
 zip_peek_cache_clear, 180