Package ‘checktor’

October 14, 2026

Title Extra CRAN Submission Checks

Version 0.2.0

Description Provides automated checks for common Comprehensive R Archive Network (CRAN) submission issues that are not caught by standard 'R CMD check'. Consolidates ad-hoc requirements that CRAN reviewers enforce but standard checks do not surface, helping R package maintainers identify and fix issues before submission to reduce rejection rates. Covers code-pattern issues, DESCRIPTION-field formatting, documentation problems, and general package structure concerns.

License AGPL (>= 3)

URL https://github.com/coatless-rpkg/checktor, https://r-pkg.thecoatlessprofessor.com/checktor/

BugReports https://github.com/coatless-rpkg/checktor/issues

Depends R (>= 4.5.0)

Imports utils, tools, cli (>= 3.0.0), generics, xml2, xmlparsedata

Suggests testthat (>= 3.2.0), withr, quarto, jsonlite, curl

VignetteBuilder quarto

Encoding UTF-8

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author James Joseph Balamuta [aut, cre, cph] (ORCID: <https://orcid.org/0000-0003-2826-8458>)

Maintainer James Joseph Balamuta <james.balamuta@gmail.com>

Repository CRAN

Date/Publication 2026-10-14 02:40:02 UTC

Contents

checktor . . . 4
checktor_category_result . . . 6
checktor_check_result . . . 7
checkup . . . 9
ci_report . . . 10
collect_rd_text . . . 11
configure_doctor . . . 12
diagnose_code_issues . . . 13
diagnose_description_issues . . . 14
diagnose_documentation_issues . . . 15
diagnose_general_issues . . . 16
diagnose_policy_violations . . . 17
example_diagnose_scenario . . . 18
extract_rd_section . . . 20
find_package_root . . . 21
health_report . . . 22
issues . . . 22
lab_acronyms . . . 23
lab_authors . . . 24
lab_browser_calls . . . 25
lab_citation_file . . . 26
lab_code_exercised . . . 27
lab_commented_examples . . . 28
lab_core_usage . . . 29
lab_cph_role . . . 30
lab_cran_comments_file . . . 31
lab_date_format . . . 32
lab_description_fields . . . 33
lab_description_file . . . 34
lab_description_function_quotes . . . 35
lab_description_length . . . 36
lab_description_placeholders . . . 37
lab_description_quoted_quotes . . . 38
lab_description_starts_with . . . 39
lab_detect_cores_robustness . . . 40
lab_donttest_vs_dontrun . . . 41
lab_encoding_utf8 . . . 42
lab_example_installs . . . 43
lab_example_interactive . . . 44
lab_example_internal_ns . . . 45
lab_example_state . . . 46
lab_example_structure . . . 47
lab_example_tf_usage . . . 47
lab_example_unparseable . . . 48
lab_example_writes . . . 49
lab_file_operations . . . 50

lab_format_names . . . 51
lab_globalenv_mod . . . 52
lab_hardcoded_credentials . . . 53
lab_home_writing . . . 54
lab_identifier_format . . . 55
lab_installed_packages . . . 56
lab_internal_ns . . . 57
lab_language_names . . . 58
lab_library_in_pkg . . . 59
lab_license . . . 60
lab_license_file_unneeded . . . 61
lab_license_year . . . 62
lab_missing_examples . . . 63
lab_network_operations . . . 64
lab_news_file . . . 65
lab_option_changes . . . 66
lab_package_size . . . 67
lab_print_cat_usage . . . 68
lab_rd_bibliography . . . 69
lab_rd_bibliography_files . . . 70
lab_readme_links . . . 71
lab_references . . . 72
lab_roxygen_usage . . . 73
lab_seed_setting . . . 74
lab_software_install . . . 75
lab_software_names . . . 76
lab_spelling . . . 77
lab_suggested_in_examples . . . 78
lab_system_calls . . . 79
lab_sys_setenv . . . 80
lab_temp_cleanup . . . 81
lab_tf_usage . . . 82
lab_title_case . . . 83
lab_title_length . . . 84
lab_title_package_name . . . 85
lab_title_redundant_phrases . . . 86
lab_title_starts_with_article . . . 87
lab_unexported_example_ns . . . 88
lab_urls . . . 89
lab_url_liveness . . . 90
lab_value_tags . . . 91
lab_version_format . . . 92
lab_warn_option . . . 93
not_under_fn_with_call_xpath . . . 94
predicates . . . 95
prescribe . . . 97
print.checktor_category_result . . . 97
print.checktor_check_result . . . 98

print.checktor_results 99
read_r_xml 99
registered_checks 100
register_check 101
show_example_files 102
summary.checktor_category_result 103
tidy.checktor_results 104
undesirable_function_check 105
unregister_check 105
xpath_lints 106
xpath_per_file 107

Index **108**

`checktor` *Diagnose Package for CRAN Submission Issues*

Description

Runs a comprehensive diagnostic suite for common CRAN submission issues that are not caught by standard R CMD check. Like a doctor for your package, this function examines your code, DESCRIPTION file, documentation, general package structure, and CRAN policy compliance to identify potential problems that could cause CRAN submission delays or rejections.

Usage

```
checktor(
  path = ".",
  verbose = getOption("checktor.verbose", TRUE),
  progress = getOption("checktor.progress", verbose),
  severity = getOption("checktor.severity", c("policy", "robustness"))
)
```

Arguments

`path`	Character. Any directory inside the R package, or a file within one. Defaults to the working directory (`"."`). checktor walks up to find the `DESCRIPTION`, so running it from `R/` or `tests/testthat/` examines the whole package rather than failing. See `find_package_root()`.
`verbose`	Logical. Whether to print detailed diagnostic output to console. Defaults to `getOption("checktor.verbose", TRUE)`.
`progress`	Logical. Whether to show progress bars during diagnostics. Defaults to `getOption("checktor.progres` `verbose)`.
`severity`	Character. Which severity tiers count toward the verdict: any of `"policy"`, `"robustness"`, `"opinion"`. Defaults to `getOption("checktor.severity", c("policy", "robustness"))`.

Every check still runs, and every finding stays in the result and appears in issues() with its tier. What this argument decides is which findings count against a clean bill of health. "policy" is a citable CRAN Repository Policy or Writing R Extensions violation. "robustness" is a real defect that CRAN will still accept, such as a detectCores() that may return NA. "opinion" is a convention with no authority behind it.

The default therefore makes "0 issues" mean *nothing here will get you rejected, and nothing here will crash a user*. Pass all three tiers to hold yourself to the conventions as well.

Details

The function runs five categories of diagnostics: **Code**, **DESCRIPTION**, **Documentation**, **General**, and **Policy**. See diagnose_code_issues(), diagnose_description_issues(), diagnose_documentation_issues diagnose_general_issues(), and diagnose_policy_violations() for the specific checks within each category.

The metadata$total_issues figure counts the individual findings of the checks in the severity tiers (e.g., 80 lines using T/F count as 80, not 1), and metadata$failed_checks counts how many of those checks reported anything. Findings in the other tiers are still in the result and are counted in metadata$advisory_issues instead.

A few checks never join a run, because no authority backs them or they ask about a submission workflow rather than the package. They are not skipped, since nothing tried to run them. metadata$on_request_checks names them; call lab_<name>() to run one.

A package can configure checktor from Config/checktor/* fields in its own DESCRIPTION (comma-separated lists; DESCRIPTION has no comment syntax, so anything after the last name becomes part of it):

- Config/checktor/disable: check names to skip entirely. A disabled check does not run and is not counted anywhere in the results. To turn a check off in every package, set options(checktor.disable = "news_file") once, for example in .Rprofile; the two lists are combined.
- Config/checktor/allow: check to mute a whole check, or check:substring to mute only findings whose text contains substring. The check still runs; muted findings are dropped from the results and tallied in metadata$suppressed, while a disabled check is removed entirely and never counted there.
- Config/checktor/software_names, Config/checktor/language_names, Config/checktor/format_names, Config/checktor/acronyms: names appended to those checks' vocabularies.

Value

A checktor_results object (list) containing:

- code_issues: Results from code diagnostics
- description_issues: Results from DESCRIPTION file diagnostics
- documentation_issues: Results from documentation diagnostics
- general_issues: Results from general package diagnostics
- policy_issues: Results from CRAN policy violation diagnostics

- metadata: List with the package path, the diagnosis time, the issue and failed-check counts (total_issues, failed_checks), the severity tiers they count, the findings outside those tiers (advisory_issues), the findings muted by Config/checktor/allow (suppressed), the checks that did not run (skipped_checks), the checks that run only when called (on_request_checks), and the checktor version (checktor_version)

Each diagnostic category contains a passed element showing which individual checks did not fail, plus detailed results for each check. A check that did not run is TRUE there, and its own skipped element records that it did not run.

Options

- checktor.verbose, checktor.progress: the defaults for verbose and progress. configure_doctor() sets both.
- checktor.severity: the default tiers for checktor() and checkup().
- checktor.disable: checks to leave out of every run, combined with Config/checktor/disable.
- checktor.url_check: TRUE or FALSE to run or skip lab_url_liveness() wherever you are. It defaults to interactive().
- checktor.spelling: FALSE turns off lab_spelling(), which is then reported as skipped.

See Also

health_report() to generate detailed reports, prescribe() for treatment recommendations, checkup() for quick health checks

Examples

```
# Run against a synthetic package with known T/F issues
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)

results              # the diagnosis summary
summary(results)     # per-category overview
issues(results)      # every issue as a tidy data frame
is_healthy(results)  # FALSE
```

---

checktor_category_result

Create a Multi-Category Diagnostic Result Object

Description

Constructor function for creating diagnostic category result objects used by multi-category diagnostic functions like diagnose_code_issues().

Usage

```
checktor_category_result(...)
```

Arguments

...	Named arguments where each is a checktor_check_result object representing individual checks within the category.

Value

An object of class `checktor_category_result` containing:

- Individual checktor_check_result objects for each check
- `passed`: Named logical vector showing which individual checks did not fail. A check that did not run is `TRUE` here, since it cannot fail a verdict; its own `skipped` element records that it did not run.

See Also

Multi-category functions like `diagnose_code_issues()`, `diagnose_documentation_issues()`

Examples

```
# Create individual check results
tf_check <- checktor_check_result(FALSE, "file.R:5", "T/F usage check")
seed_check <- checktor_check_result(TRUE, character(0), "Seed setting check")

# Create category result
code_results <- checktor_category_result(
  tf_usage = tf_check,
  seed_setting = seed_check
)
print(code_results)
```

---

checktor_check_result *Create a Standard Diagnostic Check Result Object*

Description

Constructor function for creating consistent diagnostic check result objects used by all individual diagnostic functions.

Usage

```
checktor_check_result(passed, issues, message, ...)
```

Arguments

passed	Logical. TRUE if the check passed, FALSE if issues were found.
issues	Character vector. Specific issues found, typically in "file:line" format.
message	Character. Description of what was checked.
...	Additional named elements specific to the particular check. Two are read by checktor itself: skipped = TRUE marks a check that could not run, and skip_reason says why. See Details.

Details

A check that could not run where it is, because it needs a network, a tool or a file that is not there, should say so rather than pass. Return checktor_check_result(TRUE, character(0), "<message>", skipped = TRUE, skip_reason = "<why>"). passed stays TRUE so the skip cannot fail a verdict, and skipped makes tidy(), summary(), metadata$skipped_checks, the printed report, health_report() and ci_report() report it as a check that did not run rather than one that passed. A result with passed = FALSE reads as failed whatever skipped says.

Value

An object of class checktor_check_result containing:

- passed: The passed status
- issues: Vector of issues found
- message: Description of the check
- Additional elements passed via ...

See Also

Individual diagnostic functions like lab_tf_usage(), lab_seed_setting()

Examples

```
# Create a passing check result
result <- checktor_check_result(
  passed = TRUE,
  issues = character(0),
  message = "Example check"
)
print(result)

# Create a failing check result with additional elements
result <- checktor_check_result(
  passed = FALSE,
  issues = c("file1.R:5", "file2.R:10"),
  message = "T/F usage check",
  file_issues = list("file1.R" = 5, "file2.R" = 10)
)
print(result)
```

```
# Report a check that could not run where it is
result <- checktor_check_result(
  passed = TRUE,
  issues = character(0),
  message = "License server check",
  skipped = TRUE,
  skip_reason = "no license server configured"
)
print(result)
```

---

checkup *Quick Health Check*

Description

Runs `checktor()` with minimal output, suitable for CI/CD pipelines.

Usage

```
checkup(
  path = ".",
  severity = getOption("checktor.severity", c("policy", "robustness"))
)
```

Arguments

`path`	Character. Path to the R package directory. Default: `".."`.
`severity`	Character. Which severity tiers count toward the result: any of `"policy"`, `"robustness"`, `"opinion"`. Defaults to `getOption("checktor.severity", c("policy", "robustness"))`, so a build is not failed by a convention nobody enforces. Pass all three to hold the package to the conventions as well. See `checktor()`.

Value

Logical. `TRUE` if no finding in the `severity` tiers was found, `FALSE` otherwise. Findings in the other tiers do not change it.

Examples

```
# A package that uses a bare T fails the gate
pkg_bad <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                     show_content = FALSE)
checkup(pkg_bad)
```

ci_report *Report Findings in the Format Your CI Understands*

Description

Renders a checktor() result as machine-readable findings, so a build puts each one next to the line that caused it instead of leaving it in a log for someone to read. The format defaults to whichever forge the build is running on.

Usage

```
ci_report(
  results = NULL,
  format = c("auto", "github", "gitlab", "checkstyle", "sarif", "azure", "text"),
  file = NULL,
  severity = c("policy", "robustness", "opinion"),
  skipped = TRUE,
  path = "."
)
```

Arguments

results	A checktor_results object. Defaults to running checktor() on path quietly, so a build can call this on its own.
format	Character. One of "auto", "github", "gitlab", "checkstyle", "sarif", "azure" or "text". "auto" reads the environment variables each forge sets: "github" when GITHUB_ACTIONS is set, "gitlab" for GITLAB_CI, "azure" for TF_BUILD, "checkstyle" for JENKINS_URL, and "text" anywhere else.
file	Character. Where to write. Leave it out for the format's natural destination: standard output for "github", "azure" and "text", and gl-code-quality-report.json, checktor-checkstyle.xml or checktor.sarif for the report styles. Passing file = NULL is not the same as leaving it out: nothing is written and the lines are only returned, which is what you want when testing or post-processing a report rather than handing it to a build. Pass stdout() to print regardless of format.
severity	Character. Which tiers to report. Defaults to every tier, since an annotation is information rather than a verdict.
skipped	Logical. Report the checks that did not run. Defaults to TRUE, so a green pipeline never implies a check that never happened. Named once in aggregate rather than one annotation per check.
path	Character. The package to examine when results is not supplied.

Details

Each forge reads a different shape, and format picks it:

- "github" writes workflow commands to standard output, which GitHub Actions turns into annotations on the pull request diff. Gitea and Forgejo Actions read the same commands.
- "gitlab" writes a Code Quality report, which GitLab shows on the merge request diff. Name the file in artifacts:reports:codequality:.
- "checkstyle" writes Checkstyle XML, which Jenkins, reviewdog and most review bots read. This is the one to reach for on a forge with no format of its own.
- "sarif" writes SARIF 2.1.0, which GitHub code scanning and Azure ingest.
- "azure" writes Azure Pipelines logging commands.
- "text" writes one plain line per finding, for a build with no forge at all.

Findings carry a file name rather than a path, so the path is recovered by looking for the file under R/, man/, vignettes/ and the other places a package keeps code. A finding that names a file and no line, such as a help page with no \value, is reported on that file's first line. A finding with no location at all, such as a DESCRIPTION field problem, is reported against DESCRIPTION so it still appears.

Value

Invisibly, the character vector written.

See Also

checkup() for the pass or fail gate, health_report() for a report a person reads.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)

# What a GitHub Actions job would emit. Indented here, because a runner reads
# any line starting with `::` as a command, including one this example printed.
writeLines(paste0("  ", head(ci_report(results, format = "github", file = NULL), 3)))
```

---

collect_rd_text *Flatten a Parsed* .Rd *Node to Text*

Description

Recursively concatenates the text of a tools::parse_Rd() node, optionally skipping subsections by Rd_tag (for instance "\\dontrun" when collecting example code that is meant to run).

Usage

```
collect_rd_text(node, skip = character(0))
```

Arguments

node	An Rd node, such as one returned by extract_rd_section().
skip	Character vector of Rd_tag values to omit from the text.

Value

A single character string.

See Also

extract_rd_section().

Examples

```
rd_file <- tempfile(fileext = ".Rd")
writeLines(c("\\name{foo}", "\\title{Foo}",
             "\\examples{ f() \\dontrun{ g() } }"), rd_file)
rd <- tools::parse_Rd(rd_file)
collect_rd_text(extract_rd_section(rd, "\\examples"), skip = "\\dontrun")
```

---

configure_doctor *Configure Package Doctor Defaults*

Description

Sets session-wide defaults for checktor() behavior. Subsequent calls to checktor() (and helpers that delegate to it) pick up these defaults via getOption().

Usage

```
configure_doctor(verbose_default = TRUE, progress_default = TRUE, color = TRUE)
```

Arguments

verbose_default	Logical. Default verbosity for checktor().
progress_default	Logical. Default progress-bar setting.
color	Logical. Whether cli should emit ANSI color. Sets cli.num_colors via options().

Value

Invisibly returns the previous values of the changed options, so the call can be reversed with options(.).

Examples

```
# configure_doctor() returns the options it replaced
old <- configure_doctor(verbose_default = FALSE)
getOption("checktor.verbose")

# Put them back
options(old)
```

---

diagnose_code_issues *Diagnose Code Health Issues*

Description

Runs comprehensive diagnostics on R source code to identify common CRAN submission issues and coding best-practice violations.

Usage

```
diagnose_code_issues(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to the R package directory. Default: `".."`.
`verbose`	Logical. Whether to print detailed diagnostic output. Default: `TRUE`.

Details

Each source file is parsed once with `parse(keep.source = TRUE)`; checks run XPath queries against the parsed XML representation, so identifiers that appear only inside string literals or comments do not false-positive. Multi-line constructs (`set.seed(\n123\n)`), formula `~` versus path `~`, and scope-aware patterns (an `options()` call guarded by a sibling `on.exit()` in the same function body) are all handled correctly.

Value

List of named `checktor_check_result()` objects (e.g., `tf_usage`, `seed_setting`) plus a `passed` named logical vector summarizing pass/fail for each sub-check.

See Also

`checktor()` for complete package diagnostics

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
code_results <- diagnose_code_issues(pkg, verbose = FALSE)
summary(code_results)   # per-category overview
issues(code_results)    # the issues found
```

diagnose_description_issues

Diagnose DESCRIPTION File Issues

Description

Runs diagnostics against the package DESCRIPTION file. Fields are parsed with `base::read.dcf()` so that multi-line fields like `Description` and `Title` are inspected in full, not just their first physical line.

Usage

```
diagnose_description_issues(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to the R package directory. Default: `".."`.
`verbose`	Logical. Whether to print diagnostic output. Default: `TRUE`.

Value

List containing one named element per check. Each element is a list with at least `passed`, `issues`, and `message` (see `checktor_check_result()`). When the `DESCRIPTION` is missing or R cannot read it, `description_file` fails, and every check that reads the parsed fields, registered ones included, is reported as skipped with the reason "DESCRIPTION could not be read", since it has nothing to examine.

See Also

`checktor()` for complete package diagnostics

Examples

```
pkg_path <- example_diagnose_scenario("description_examples/bad_description.txt",
                                      show_content = FALSE)
results <- diagnose_description_issues(pkg_path, verbose = FALSE)
issues(results)     # description-field problems, if any
```

diagnose_documentation_issues

Diagnose Documentation Issues

Description

Runs diagnostics on package documentation to identify common issues that can cause CRAN submission problems or a poor user experience.

Usage

```
diagnose_documentation_issues(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to package directory. Default: `"."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.

Details

This function checks for:

- Missing `\value` tags in function documentation
- Exported functions missing an `\examples` section
- Roxygen2 usage
- Example structure (appropriate use of `\dontrun{}`)
- Examples that use Suggested packages without a guard

`.Rd` files are parsed structurally via `tools::parse_Rd()` so analyses look at sections by their `Rd_tag` rather than grepping LaTeX text.

Value

List of `checktor_check_result()` objects plus a `passed` named logical vector summarizing pass/fail per check.

See Also

`checktor()` for complete package diagnostics

Examples

```
pkg_path <- example_diagnose_scenario("documentation_examples/missing_value_tag.Rd",
                                      show_content = FALSE)
doc_results <- diagnose_documentation_issues(pkg_path, verbose = FALSE)
summary(doc_results)
issues(doc_results)
```

diagnose_general_issues

Diagnose General Package Issues

Description

Runs general diagnostics on package structure and content that don't fit into specific code, documentation, or DESCRIPTION categories.

Usage

```
diagnose_general_issues(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory. Default: `"."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.

Details

This function checks:

- Package size, measured against the files that would go into the tarball (`.Rbuildignore` and standard scratch dirs are excluded), with a 5 MB warning threshold matching CRAN's recommendation.
- `http://` URLs and URL shorteners, offline. `R CMD check --as-cran` does fetch every URL, but only with a network and only under `--as-cran`; this is the fast local pass.
- Presence of a `NEWS` file documenting user-facing changes.
- Relative links in the `README` that would break on CRAN.
- A package that exports code but has no examples, no tests and no vignettes to exercise it.
- An `inst/CITATION` that uses the old-style `citEntry()` or `personList()`, or calls `packageDescription()`, `library()` or `require()`.

`lab_cran_comments_file()` is intentionally not part of this default run, since a `cran-comments.md` is a workflow convention rather than a CRAN requirement; call it directly to opt in.

Value

List of `checktor_check_result()` objects plus a `passed` summary.

See Also

`checktor()` for complete package diagnostics

Examples

```
pkg_path <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                      show_content = FALSE)
general_results <- diagnose_general_issues(pkg_path, verbose = FALSE)
general_results$package_size$size_mb
```

---

diagnose_policy_violations

Check for Common CRAN Policy Violations

Description

Runs additional diagnostics focused on CRAN policy: leftover `browser()` calls, raw system invocations, file writes outside `tempdir()`, and unwrapped network access in examples or vignettes. Code-side checks use the parsed AST so string/comment matches don't false-positive; Rd-side checks use `tools::parse_Rd()` for the same reason.

Usage

```
diagnose_policy_violations(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to the R package directory. Default: `".."`.
`verbose`	Logical. Whether to print diagnostic output. Default: `TRUE`.

Value

List of `checktor_check_result()` objects, plus a `passed` named logical vector summarizing pass/fail per check.

See Also

`checktor()` for complete package diagnostics

Examples

```
pkg <- example_diagnose_scenario("code_examples/browser_calls_bad.R",
                                 show_content = FALSE)
policy <- diagnose_policy_violations(pkg, verbose = FALSE)
summary(policy)
issues(policy)
```

example_diagnose_scenario

Create Example Diagnostic Scenario

Description

Creates a temporary package structure with a specified example file for testing diagnostic functions. This is primarily used in documentation examples to demonstrate diagnostic capabilities with known problematic code.

Usage

```
example_diagnose_scenario(
  example_path,
  show_content = TRUE,
  description_type = "minimal",
  cleanup = FALSE
)
```

Arguments

`example_path` Character. Relative path to example file within inst/diagnose/. Should include subdirectory and filename (e.g., "code_examples/tf_usage_bad.R").

`show_content` Logical. Whether to display the example file content in the console. Default: `TRUE`.

`description_type` Character. Type of DESCRIPTION file to create. Options: "minimal" (basic fields only), "bad" (with known issues), "good" (properly formatted). Default: "minimal". Ignored when `example_path` is a `.txt` scenario, which becomes the `DESCRIPTION` itself.

`cleanup` Logical. Whether to delete the temporary package when the function that called `example_diagnose_scenario()` returns. Called at top level there is no such function, so the package stays until R removes the session's temporary directory. Default: `FALSE` (you manage cleanup).

Details

This function:

1. Locates the specified example file in the package's `inst/diagnose/` directory
2. Creates a temporary package directory structure
3. Copies the example file to where a package keeps its kind: an `.R` file in `R/`, an `.Rd` file in `man/`, a vignette (`.Rmd`, `.qmd`, `.Rnw`) in `vignettes/`, a DESCRIPTION scenario (`.txt`) as the `DESCRIPTION`, a citation scenario (`.CITATION`) as `inst/CITATION`, and a licence scenario (`.LICENSE`) as `LICENSE`
4. Optionally displays the example file content

5. Returns the path to the temporary package for diagnostic testing

The temporary package includes minimal structure (`R/`, `man/`, etc.) needed for running diagnostics, plus a basic `DESCRIPTION` file.

Value

Character. Path to the temporary package directory containing the example file. Returns `NULL`, with a warning, if the example file cannot be found. An `example_path` that is not an `.R`, `.Rd`, `.Rmd`, `.qmd`, `.Rnw`, `.txt`, `.CITATION` or `.LICENSE` file is an error, since a package has no place for it.

Example File Structure

The temporary package created has this structure. The example file goes in the one place its extension names, and the other directories stay empty:

```
<tempdir>/checktor_example_XXXX/
|-- DESCRIPTION          # The template, or a .txt scenario itself
|-- LICENSE              # A .LICENSE scenario; made only for one
|-- NEWS.md              # So the NEWS check has nothing to report
|-- cran-comments.md     # So the cran-comments check has nothing to report
|-- R/                   # An .R scenario, such as tf_usage_bad.R
|-- man/                 # An .Rd scenario, such as missing_value_tag.Rd
|-- inst/CITATION        # A .CITATION scenario; made only for one
|-- tests/               # Always empty
`-- vignettes/           # An .Rmd, .qmd or .Rnw scenario; made only for one
```

See Also

Used in examples for diagnostic functions like `lab_tf_usage()`, `lab_seed_setting()`, etc.

Examples

```
# A scenario with T/F usage issues. show_content defaults to TRUE, so the
# offending file prints first
pkg_path <- example_diagnose_scenario("code_examples/tf_usage_bad.R")
lab_tf_usage(pkg_path, verbose = FALSE)$issues
issues(checktor(pkg_path, verbose = FALSE, progress = FALSE))
unlink(pkg_path, recursive = TRUE)

# A .txt scenario is the DESCRIPTION, so it needs no description_type
pkg_path <- example_diagnose_scenario("description_examples/bad_description.txt",
                                      show_content = FALSE)
issues(diagnose_description_issues(pkg_path, verbose = FALSE))
unlink(pkg_path, recursive = TRUE)

# With cleanup = TRUE the package is deleted when the calling function returns
count_tf <- function() {
  pkg_path <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                        show_content = FALSE, cleanup = TRUE)
  length(lab_tf_usage(pkg_path, verbose = FALSE)$issues)
```

```
}
count_tf()
```

`extract_rd_section` *Extract One Section from a Parsed* `.Rd` *File*

Description

Returns the first top-level node of a `tools::parse_Rd()` result whose `Rd_tag` matches `tag`, or `NULL` if absent. Use it to reach a specific section, such as `\\value` or `\\examples`, when writing a documentation check.

Usage

```
extract_rd_section(rd, tag)
```

Arguments

`rd`	A parsed `.Rd` object from `tools::parse_Rd()`.
`tag`	Character. The `Rd_tag` to find, e.g. `"\\value"` or `"\\examples"`.

Value

The matching Rd node, or `NULL`.

See Also

`collect_rd_text()`.

Examples

```
rd_file <- tempfile(fileext = ".Rd")
writeLines(c("\\name{foo}", "\\title{Foo}", "\\value{A number.}"), rd_file)
rd <- tools::parse_Rd(rd_file)
collect_rd_text(extract_rd_section(rd, "\\value"))
```

find_package_root — *Find the Root of the Package Containing a Path*

Description

Walks up from `path` until it finds the directory holding a `DESCRIPTION` file, so checktor can be run from anywhere inside a package tree rather than only from the directory holding `DESCRIPTION`. That is what lets `checktor()` work with your working directory set to `R/`, `tests/testthat/`, or any other subdirectory.

Usage

```
find_package_root(path = ".")
```

Arguments

`path`	Character. A directory inside a package, or a file within one. Default: `".."`.

Details

Every checktor entry point calls this on the `path` it is given, so you rarely need it directly. It is exported for custom checks registered with `register_check()`, which receive a path that has already been resolved.

Value

Character. The package root, as an absolute path, when one is found above `path`. When `path` itself holds a `DESCRIPTION` it is returned unchanged, so an existing caller sees exactly what it passed in. When no package root is found at all, the search falls back to the directory it started from, meaning `path` for a directory and its parent for a file, which leaves the caller reporting the problem against the place the user pointed at.

See Also

`checktor()`, which resolves its `path` this way.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)

# From the package root, the path is handed back untouched
identical(find_package_root(pkg), pkg)

# From a subdirectory, the root is found by walking up
basename(find_package_root(file.path(pkg, "R")))
```

health_report — *Comprehensive Health Report*

Description

Creates a report of every failing check, whatever its severity tier, with the treatment `prescribe()` gives for it. Every format carries the same treatment text.

Usage

```
health_report(results, file = NULL, format = "markdown")
```

Arguments

`results`	A `checktor_results` object from `checktor()`.
`file`	Character. A path to write the report to, or `NULL` (the default) to only return it.
`format`	Character. Report format: `"markdown"` (the default), `"html"`, or `"text"`. Any other value gives the text format.

Value

The report as a character vector, one element per line. It is returned visibly even when `file` is given, so assign it to keep a console call from printing it.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)
report <- health_report(results, format = "text")
head(report)
```

issues — *Extract issues, checks, or a per-category summary from checktor results*

Description

Plain accessors over the objects returned by `checktor()` and the `diagnose_*_issues()` functions, so you never navigate nested sublists.

Usage

```
issues(x, ...)

## S3 method for class 'checktor_check_result'
issues(x, ...)

## S3 method for class 'checktor_category_result'
issues(x, ...)

## S3 method for class 'checktor_results'
issues(x, ...)
```

Arguments

x	A `checktor_results`, `checktor_category_result`, or `checktor_check_result` object.
...	Unused.

Value

`issues()` returns a `data.frame` with one row per issue, whatever its severity tier. At the results level the columns are `category`, `check`, `severity`, `file`, `line`, `location`, `message`; a single category drops `category`; a single check drops `category`, `check` and `severity`. A healthy object yields a 0-row frame. `file` and `line` are read from `location`: a finding that names a file and no line, such as a help page with no `\value`, has the file and an `NA` line, and a finding that names no file, such as one about a `DESCRIPTION` field, has `NA` for both.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)
issues(results)
```

---

lab_acronyms *Diagnose Unexplained Acronyms in DESCRIPTION*

Description

Flags an acronym in `Description` that is never spelled out. A parenthetical gloss in either order counts as explained.

Usage

```
lab_acronyms(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: `"."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.
desc	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Details

A name in single quotes, straight or typographic, is not read as an acronym, so `'YAML'` or `'MATLAB'` written as `lab_language_names()` and `lab_format_names()` ask is not reported here. Nor is anything inside a web address, a `<doi:...>`, a function call or a quotation in double quotes, straight or typographic, such as the title of an article.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Explaining Acronyms. Reviewers ask for an acronym to be spelled out once, but nothing enforces it, which is why this sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/acronyms_bad.txt",
                                 show_content = FALSE)
lab_acronyms(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_authors *Diagnose the Authors@R Field*

Description

Flags a missing `Authors@R`, and an unfilled `usethis` or `package.skeleton()` template such as `person("First", "Last", ...)` or `person("Givenname", "Familyname", ...)`, which is a hard CRAN rejection that `R CMD check` says nothing about.

Usage

```
lab_authors(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: `".."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.
desc	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Details

The field is evaluated without running any code it contains: only `person()`, `as.person()`, `c()`, `list()`, `paste()`, `paste0()` and `(` resolve, the calls R's own reader allows from R 4.6.0 on. Every other call in the field is reported, since `R CMD build` refuses it as a malformed `Authors@R` field. That includes a namespaced `utils::person()` and a function in parentheses, as in `(c)(...)`, which are still read so the other checks see their roles.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Repository Policy treats a placeholder or malformed `Authors@R`, including a missing maintainer, as a rejection. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/authors_bad.txt",
                                 show_content = FALSE)
lab_authors(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_browser_calls *Diagnose Leftover browser() Calls*

Description

Flags a `browser()` call left in package code.

Usage

```
lab_browser_calls(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

path	Character. Path to the package directory. Default: `".."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.
parsed	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Repository Policy requires checks to run non-interactively, so a debugging leftover such as `browser()` must not be left in. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/browser_calls_bad.R",
                                 show_content = FALSE)
lab_browser_calls(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_citation_file *Diagnose Old-Style or Unsafe Calls in inst/CITATION*

Description

Flags an `inst/CITATION` that calls the old-style `citEntry()` (use `bibentry()`) or `personList()` and `as.personList()` (use `c()` on `person` objects), or that calls `packageDescription()`, `library()` or `require()`. R passes the file a `meta` object holding the package's DESCRIPTION, so it needs none of those, and each assumes the package is already installed.

Usage

```
lab_citation_file(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to package directory
verbose	Logical. Print diagnostic messages

Details

As in R, the fallback `if (!exists("meta") || is.null(meta)) meta <- packageDescription("pkg")` is exempt: a top-level `if` with exactly that condition and no `else`. The file is parsed and never run, so a name in a comment or a string is not a call. A CITATION that does not parse is reported, with the line R stops at. A CITATION that `.Rbuildignore` excludes is not in the tarball and is not read.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN incoming check (`R CMD check --as-cran`) NOTEs "Package CITATION file contains call(s) to old-style citEntry(). Please use bibentry() instead." and "... old-style personList() or as.personList(). Please use c() on person objects instead.", and lists calls to `packageDescription`, `library` and `require`. Writing R Extensions, under "CITATION files", says "It is desirable (and essential for CRAN) that the CITATION file does not contain calls to functions such as packageDescription which assume the package is installed in a library tree on the package search path." `devtools::check()` turns the incoming check off by default. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("general_examples/citation_file_bad.CITATION",
                                 show_content = FALSE)
lab_citation_file(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_code_exercised *Diagnose a Package Nothing Exercises*

Description

Flags a package that exports something but ships no `\examples` in any help page, no tests and no vignettes, so a check runs none of its code. CRAN's incoming check raises this as a WARNING, which is an automatic rejection.

Usage

```
lab_code_exercised(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Details

The conditions are R's own. An example counts even when all of it sits in `\dontrun{}`, because R still writes it out. A test counts only as a file directly in `tests/` ending in `.R`, `.r` or `.Rin`, since that is what `R CMD check` runs: a `tests/testthat/` folder without the `tests/testthat.R` driver is never run, and the finding says so. Files `.Rbuildignore` excludes do not count, since they are not in the tarball. A package with no `R/` directory, or whose `NAMESPACE` exports nothing, passes. So does one whose `NAMESPACE` is missing or cannot be parsed, since its exports are then unknown.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN incoming check (`R CMD check --as-cran`) reports "checking for code which exercises the package ... WARNING" with "No examples, no tests, no vignettes" when all three are missing and the `NAMESPACE` exports anything. `devtools::check()` turns the incoming check off by default, so the WARNING is usually first seen at win-builder or on submission. The CRAN Repository Policy asks that "the checks that are left do exercise all the features of the package". See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("general_examples/code_exercised_bad.R",
                                 show_content = FALSE)
# The scenario's function is exported, as roxygen2 would record it
writeLines("export(tidy_values)", file.path(pkg, "NAMESPACE"))
lab_code_exercised(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_commented_examples

Diagnose Examples That Run Nothing

Description

Flags an `\examples{}` block whose only content is commented out, so it demonstrates nothing. A comment beside live code is illustration and is not flagged.

Usage

```
lab_commented_examples(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.

Value

checktor_check_result() with passed, issues, message.

Source

No formal rule. An \examples{} block that is entirely commented out demonstrates nothing, a convention which is why this sits at opinion tier. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/commented_examples_bad.Rd",
                                 show_content = FALSE)
lab_commented_examples(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_core_usage *Diagnose Parallel Core Usage*

Description

Flags a worker count that can exceed CRAN's two-core limit. Understands parallel, snow, foreach, future, furrr, mirai, RcppParallel, data.table, and BiocParallel.

Usage

```
lab_core_usage(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ". ".
verbose	Logical. Print diagnostic output. Default: TRUE.
parsed	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing path afresh.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Repository Policy states that "If running a package uses multiple threads/cores it must never use more than two simultaneously". See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/core_usage_bad.R",
                                 show_content = FALSE)
lab_core_usage(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_cph_role *Diagnose a Missing Copyright Holder*

Description

Flags a package that names no copyright holder: no person in Authors@R has the cph role and there is no Copyright field. It runs only when you call it: authors who are natural persons hold copyright by default, so most packages need neither. It is worth running when an organisation owns the copyright, since that is the case the role exists for.

Usage

```
lab_cph_role(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.
desc	Optional pre-parsed DESCRIPTION, as returned by base::read.dcf(). Defaults to reading it from path.

Details

The roles are read from the parsed person() object, never from the text of the field, so an address such as cph@example.com is not a role. The field is parsed without running any code it contains, as in lab_authors(). A package with no Authors@R is read from its legacy Author field instead, where only a role written in square brackets, as in ACME Corporation [cph], counts, in any case, and not one inside a parenthesised comment.

Value

checktor_check_result() with passed, issues, message.

Source

utils::person() documents "cph" as the role for "all copyright holders", adding that "authors which are 'natural persons' are by default copyright holders and so do not need to be given this role". CRAN Repository Policy asks only that copyright ownership be clear, which a Copyright field also satisfies. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/cph_role_bad.txt",
                                 show_content = FALSE)
lab_cph_role(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_cran_comments_file

Diagnose a Missing cran-comments.md File

Description

A cran-comments.md file carries the submission notes CRAN reviewers read (test environments, R CMD check results, downstream-dependency notes). Its absence is flagged so it can be added before submission.

Usage

```
lab_cran_comments_file(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to package directory
verbose	Logical. Print diagnostic messages

Details

This check is opt-in: it is **not** part of the default checktor() / diagnose_general_issues() run, because a cran-comments.md is a workflow convention rather than a CRAN requirement. Call it directly to use it.

Value

checktor_check_result() with passed, issues, message.

Source

No formal rule. A `cran-comments.md` is a submission-workflow convention rather than a CRAN requirement, which is why this check is opt-in and not part of a default run. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
invisible(file.remove(file.path(pkg, "cran-comments.md")))
lab_cran_comments_file(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_date_format *Diagnose the DESCRIPTION Date Field*

Description

Flags a `Date` field that is not ISO 8601 `yyyy-mm-dd`, is over a month old, or lies in the future. Mirrors the CRAN incoming check; an absent `Date` field (the common, preferred case) passes.

Usage

```
lab_date_format(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "The DESCRIPTION file", says "the 'yyyy-mm-dd' format of the ISO 8601 standard is strongly recommended"; the incoming check also flags a stale or future date. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/date_format_bad.txt",
                                 show_content = FALSE)
lab_date_format(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_description_fields

Diagnose DESCRIPTION Fields R Does Not Know

Description

Flags a DESCRIPTION field that is not one of the fields R knows, which is how CRAN's incoming check reads it. The usual one is Remotes, which devtools and pak read but CRAN does not, since CRAN installs dependencies from CRAN and Bioconductor alone. The others are typos such as Bugreports, Import, Suggest or URLs, which R ignores, so the field meant is silently missing; the finding names the field that was probably meant.

Usage

```
lab_description_fields(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ". ".
verbose	Logical. Print diagnostic output. Default: TRUE.
desc	Optional pre-parsed DESCRIPTION, as returned by base::read.dcf(). Defaults to reading it from path.

Details

The fields R allows beyond its own list are allowed here too: any Config/ field, such as Config/testthat/edition, and fields starting X-CRAN, X-schema.org, Repository/R-Forge or VCS/, and a standard field name followed by Note, such as RoxygenNote.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN incoming check run by R CMD check --as-cran NOTEs "Unknown, possibly misspelled, fields in DESCRIPTION". The CRAN Repository Policy says "The strong dependencies ... should be available from CRAN or the Bioconductor software repository", which is why a Remotes field has no place in a submission. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/description_fields_bad.txt",
                                 show_content = FALSE)
lab_description_fields(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_description_file *Diagnose a DESCRIPTION File R Cannot Read*

Description

Flags a DESCRIPTION that is missing or that R cannot read: a line that is neither a Field: value pair nor an indented continuation, or a blank line that splits the file in two. R CMD build and R CMD INSTALL both stop on such a file, so the package cannot be built or installed. A DESCRIPTION R cannot open at all fails with the reason, a directory in its place or a file without read permission, which is found from the file system and so reads the same in any language. Every other DESCRIPTION check reads the parsed fields, so while this one fails they are reported as skipped.

Usage

```
lab_description_file(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.
desc	Present for signature parity with the other DESCRIPTION checks; this check asks whether R can read the DESCRIPTION file itself, so it reads the file and ignores desc.

Value

checktor_check_result() with passed, issues, message.

Source

Writing R Extensions, under "The DESCRIPTION file", specifies the format of a Debian Control File: "Fields start with an ASCII name immediately followed by a colon" and "Continuation lines ... start with a space or tab". R reads it with base::read.dcf() and refuses a file that yields more than one record. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/unparseable_description.txt",
                                 show_content = FALSE)
lab_description_file(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_description_function_quotes

Diagnose Single-Quoted Function Names

Description

Flags a single-quoted function name in Title/Description. Single quotes are reserved for software names.

Usage

```
lab_description_function_quotes(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.
desc	Optional pre-parsed DESCRIPTION, as returned by base::read.dcf(). Defaults to reading it from path.

Value

checktor_check_result() with passed, issues, message.

Source

No rule. This was an invented rule and is kept callable but off by default. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/description_function_quotes_bad.txt",
                                 show_content = FALSE)
lab_description_function_quotes(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_description_length

Diagnose Description Length

Description

Flags a `Description` of fewer than 10 words.

Usage

```
lab_description_length(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Description Length. A one-line `Description` is thin and reviewers ask for more, a convention rather than a rule, which is why this sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/description_length_bad.txt",
                                 show_content = FALSE)
lab_description_length(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_description_placeholders

Diagnose Template Text Left in DESCRIPTION

Description

Flags a `Title`, `Description`, `Author` or `Maintainer` that still holds the text a package template wrote: the `usethis` template's "What the Package Does (One Line, Title Case)" and "What the package does (one paragraph).", and `package.skeleton()`'s "What the Package Does (Short Line)", "More about what it does (maybe more than one line).", "Who wrote it" and "Who to complain to". It applies R's own tests, so it flags what CRAN flags. A template `Authors@R` is `lab_authors()`'s to report.

Usage

```
lab_description_placeholders(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN incoming check run by `R CMD check --as-cran` NOTEs "DESCRIPTION fields with placeholder content". Writing R Extensions asks for a `Title` and `Description` that describe the package. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/description_placeholders_bad.txt",
                                 show_content = FALSE)
lab_description_placeholders(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_description_quoted_quotes

Diagnose Double-Quoted Software Names

Description

Flags a software name in double quotes in `Title` or `Description`. Writing R Extensions reserves double quotes for quotations and requires single quotes for software names, so scare-quoted jargon is left alone. A lower-case span matches only a name written in lower case, such as `shiny`, so an English `"rust"` or a parameter `"r"` is not read as `Rust` or `R`. In the `Title`, where Title Case capitalises every word, a capitalised `"Bugs"` is likewise the word and not `BUGS`.

Usage

```
lab_description_quoted_quotes(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Details

The names are those `lab_software_names()` and `lab_language_names()` ask to see in single quotes, and a few more. Format names such as `JSON` or `HTML`, which CRAN accepts bare, are not among them, so one in double quotes is not reported; `lab_format_names()` reports them bare, on request.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "The DESCRIPTION file", reserves double quotes for book titles and similar; software names take single quotes. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/description_quoted_quotes_bad.txt",
                                 show_content = FALSE)
lab_description_quoted_quotes(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_description_starts_with

Diagnose the Description Opening

Description

Flags a `Description` opening with a phrase CRAN forbids ("This package..."), or one that does not begin with a capital letter.

Usage

```
lab_description_starts_with(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "The DESCRIPTION file", says "It is good practice not to start with the package name, 'This package' or similar"; the incoming check flags it. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/description_starts_with_bad.txt",
                                 show_content = FALSE)
lab_description_starts_with(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_detect_cores_robustness

Diagnose Unguarded `detectCores()`

Description

Flags a `parallel::detectCores()` call whose result is used without an `NA` guard.

Usage

```
lab_detect_cores_robustness(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to package directory.
`verbose`	Logical. Print diagnostic messages.
`parsed`	Internal. Pre-parsed source cache; if `NULL`, files are read from `path` on demand.

Details

`?detectCores` says so in as many words: *"An integer,* `NA` *if the answer is unknown"*, and R's own advice is to avoid it, *"First because it may return* `NA`*"*. `NA` then propagates silently through the arithmetic packages usually do next, and the failure surfaces far from its cause:

```
n <- parallel::detectCores() - 1   # NA - 1 is NA
if (cores > n) cores <- n         # Error: missing value where TRUE/FALSE needed
```

This is a robustness defect rather than a policy one, and it is distinct from the `core_usage` check, which asks how many cores you *use*. A package can cap itself at two cores perfectly and still crash on the machine where `detectCores()` returns `NA`.

A call is treated as guarded when its enclosing function tests for `NA` (`is.na()`), passes `na.rm = TRUE`, or supplies a fallback with `%||%`. The durable fix is `parallelly::availableCores()`, which never returns `NA` and also honours the CRAN core limit.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

No formal rule. `?detectCores` states it returns "`NA` if the answer is unknown", and the arithmetic that usually follows then crashes, which is why this sits at `robustness` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("code_examples/detect_cores_bad.R",
                                 show_content = FALSE)
lab_detect_cores_robustness(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_donttest_vs_dontrun

Diagnose dontrun Where donttest Belongs

Description

Flags \dontrun{} around code that is merely slow. \donttest{} is the right wrapper, since it still runs under --run-donttest.

Usage

```
lab_donttest_vs_dontrun(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.

Details

A slow block that uses a Suggested package without a guard is left alone: R CMD check --as-cran runs \donttest{} code, so after the move lab_suggested_in_examples() would report it. Each \dontrun{} block is judged on its own, so such a block does not hold back the advice for another that is only slow.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Cookbook covers the distinction under Structuring of Examples, where \donttest{} is the wrapper for an example that merely runs long. Nothing enforces the choice, which is why this sits at opinion tier. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/donttest_vs_dontrun_bad.Rd",
                                 show_content = FALSE)
lab_donttest_vs_dontrun(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_encoding_utf8 *Diagnose a DESCRIPTION Encoding Other Than UTF-8*

Description

Flags an `Encoding` that is anything but exactly `UTF-8`, which is the test CRAN's incoming check applies. `latin1` and `latin2` are still legal R, but CRAN NOTEs them as deprecated, and the comparison is case-sensitive, so a lower-case `utf-8` is NOTEd too. An absent `Encoding` passes.

Usage

```
lab_encoding_utf8(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "The DESCRIPTION file", says a non-ASCII DESCRIPTION "should contain an 'Encoding' field"; the incoming check run by `R CMD check --as-cran` NOTEs any value but `UTF-8` with "Package encoding '...' is deprecated. Please change to UTF-8 for non-ASCII content." See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/encoding_utf8_bad.txt",
                                 show_content = FALSE)
lab_encoding_utf8(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_example_installs *Diagnose Installs in Examples, Vignettes and Demos*

Description

Flags a call that installs a package or external software from an example, a vignette or a demo. CRAN asks maintainers not to install anything from these, because a check then has to do the install too, and the user did not ask for it.

Usage

```
lab_example_installs(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to the package directory. Default: `".."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Installing Software, and it is a rejection maintainers receive verbatim: "Please do not install packages in your functions, examples or vignette." See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_software_install()` for the same rule in `R/`.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/example_installs_bad.Rd",
                                 show_content = FALSE)
lab_example_installs(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_example_interactive

Diagnose Interactive Examples Hidden in \\dontrun{} *or* \\donttest{}

Description

Flags an interactive call, such as a shiny app, a viewer or a prompt, that an example hides in \\dontrun{} or \\donttest{} instead of guarding with `if (interactive())`.

Usage

```
lab_example_interactive(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.

Details

In \\dontrun{}, CRAN asks for the guard instead, so a reader can see that the function needs a session, not only that it does not run. In \\donttest{} the call is a failure waiting to happen: `R CMD check --as-cran` runs that code, where a prompt errors and an app waits for input until the check times out.

The example is read as parsed R, so a function named in a comment or a string is not a call, and only a guard that actually encloses the call excuses it, including roxygen's `@examplesIf interactive()`. An app that `shinyApp()` builds is reported only where it is printed, which is what runs it; kept in a variable or handed to another function, it starts nothing.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Structuring of Examples, and the rejection reads "Functions which are supposed to only run interactively (e.g. shiny) should be wrapped in if(interactive()). Please replace \dontrun with if(interactive()) if possible". `R CMD check --as-cran` has run \\donttest{} examples since R 4.0.0. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_example_structure()`.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/example_interactive_bad.Rd",
                                 show_content = FALSE)
lab_example_interactive(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_example_internal_ns

Diagnose `:::` *in Examples*

Description

Flags a `pkg:::fn()` call in an example. The triple colon reaches an unexported object, whose behaviour the author is free to change, so CRAN asks for one colon or for the object to be exported.

Usage

```
lab_example_internal_ns(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

CRAN sends this back verbatim as "Using foo:::f instead of foo::f allows access to unexported objects ... Please omit one colon", listing the `.Rd` files it appears in. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_unexported_example_ns()`.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/example_internal_ns_bad.Rd",
                                 show_content = FALSE)
lab_example_internal_ns(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_example_state *Diagnose Session State Left Changed by Examples*

Description

Flags an example, vignette or demo that changes `options()`, `par()` or the working directory without putting it back. A reader who runs the example is left with a session that behaves differently afterwards.

Usage

```
lab_example_state(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Change of Options, Graphical Parameters and Working Directory, and the rejection reads "Please always make sure to reset to user's options(), working directory or par() after you changed it in examples and vignettes and demos." See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_option_changes()` for the same rule in `R/`.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/example_state_bad.Rmd",
                                 show_content = FALSE)
lab_example_state(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_example_structure *Diagnose Example Structure*

Description

Walks \examples{} sections via tools::parse_Rd() and flags \dontrun{} subtrees that don't appear to have a justifying reason (interactive, network, credentials, long-running, etc.).

Usage

```
lab_example_structure(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to package directory
verbose	Logical. Print diagnostic messages

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Cookbook covers this under Structuring of Examples. \dontrun{} should wrap only code that genuinely cannot run inside a check, a convention rather than a rule, which is why this sits at opinion tier. See vignette("check-sources", package = "checktor") for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/example_structure_bad.Rd",
                                 show_content = FALSE)
lab_example_structure(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_example_tf_usage *Diagnose T/F Usage in Examples, Vignettes and Demos*

Description

Flags a bare T or F in an example, a vignette chunk that runs, or a demo, judged exactly as lab_tf_usage() judges R/: a T in a string, a comment, an argument name, x$T or language built by quote() is not reported.

Usage

```
lab_example_tf_usage(path = ".", verbose = TRUE, tests = FALSE)
```

Arguments

path	Character. Path to the package directory. Default: `".."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.
tests	Logical. Read `tests/` as well. Default: `FALSE`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook recipe T/F Instead of TRUE/FALSE says `T` and `F` "should not be used as variable names in your code, examples, tests or vignettes", and the reviewer's letter names the `.Rd` file: "'T' and 'F' instead of TRUE and FALSE: man/quiet.Rd: quiet(x, be_quiet = T)". No rule makes it binding, so it sits at `robustness` tier like `lab_tf_usage()`. Tests are left out unless `tests = TRUE`, since CRAN rarely reads them. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_tf_usage()` for the same rule in `R/`.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/example_tf_usage_bad.Rd",
                                 show_content = FALSE)
lab_example_tf_usage(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_example_unparseable

Diagnose Examples That Are Not Valid R

Description

Flags an `\examples{}` section whose code does not parse as R, reporting the `.Rd` file and the line the parser stopped on. The code inside `\dontrun{}` is read too. `R CMD check` writes it out as comments and never parses it, so a missing bracket or a `<your key>` placeholder there passes every check.

Usage

```
lab_example_unparseable(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to the package directory. Default: `".."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.

Value

checktor_check_result() with passed, issues, message.

Source

A rejection CRAN reviewers send verbatim: "Warning: Unexecutable code in man/make.trait.model.Rd". Writing R Extensions, under "Documenting functions", says example code outside \dontrun{} "must be executable", and that the text inside it "need not be valid R code". Reviewers run the examples with \dontrun{} included all the same, so a placeholder belongs in a string or a variable, as in key <- "<your key>". An example that is deliberately not R, such as C++ source shown for reading, can be turned off with Config/checktor/disable. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), lab_example_structure().

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/example_unparseable_bad.Rd",
                                 show_content = FALSE)
lab_example_unparseable(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_example_writes *Diagnose Writes Outside the Temporary Directory in Examples*

Description

Flags a write from an example, vignette or demo whose destination is a literal path, so it lands in the user's filespace rather than in tempdir(). A destination the caller supplies is permission and is not flagged.

Usage

```
lab_example_writes(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to the package directory. Default: ". ".
verbose	Logical. Print diagnostic output. Default: TRUE.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Cookbook covers this under Writing Files and Directories to the Home Filespace, and the rejection reads "Please ensure that your functions do not write by default or in your examples/vignettes/tests in the user's home filespace". See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_file_operations()` for the same rule in `R/`.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/example_writes_bad.Rd",
                                 show_content = FALSE)
lab_example_writes(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_file_operations *Diagnose Writes to the User's Filespace*

Description

Flags a write whose destination is a literal path, so it provably lands in the working directory or the user's home. A caller-supplied or computed destination is permission, and is not flagged.

Usage

```
lab_file_operations(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`parsed`	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Repository Policy states that "Packages should not write ... anywhere ... apart from the R session's temporary directory". See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/file_operations_bad.R",
                                 show_content = FALSE)
lab_file_operations(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_format_names *Diagnose Bare Format and Markup Names in DESCRIPTION*

Description

Flags a data, markup, typesetting or query format name – JSON, HTML, XML, CSS, YAML, TOML, Markdown, LaTeX, TeX or SQL – or one of the languages CRAN packages write either way (C++, Fortran, Tcl) that appears in Title or Description without single quotes. It runs only when you call it, for a maintainer who wants the quoting consistent. CRAN accepts these names bare, so a finding here never counts against a clean result.

Usage

```
lab_format_names(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ". ".
verbose	Logical. Print diagnostic output. Default: TRUE.
desc	Optional pre-parsed DESCRIPTION, as returned by base::read.dcf(). Defaults to reading it from path.

Details

Occurrences are judged as in lab_software_names(): a name inside a quoted longer name such as 'R Markdown' or 'JSON-stat' is quoted, and so is one in a web address or a double-quoted span. A format name alone in double quotes, such as "JSON", is therefore reported by no check, by design: double quotes are the wrong kind for a name, but CRAN accepts the name with no quotes at all, so lab_description_quoted_quotes() leaves it alone too. A package can extend the list through Config/checktor/format_names in its own DESCRIPTION.

Value

checktor_check_result() with passed, issues, message.

Source

Writing R Extensions asks for single quotes around "other packages and external software", and a format is not software. A census of CRAN in September 2026 found that packages accepted at new-package review in the previous 18 months wrote these names bare 62% of the time, against 28% for the languages `lab_language_names()` covers, so this is a matter of style and sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_language_names()`.

Examples

```
pkg <- example_diagnose_scenario("description_examples/format_names_bad.txt",
                                 show_content = FALSE)
lab_format_names(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_globalenv_mod *Diagnose Writes to the Global Environment*

Description

Flags a `<<-` whose target binds in neither an enclosing function nor the package, and so reaches `.GlobalEnv`. A closure updating its parent, or a package-level cache, is not flagged.

Usage

```
lab_globalenv_mod(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`parsed`	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Repository Policy states that "Packages should not modify the global environment (user's workspace)." See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/globalenv_bad.R",
                                 show_content = FALSE)
lab_globalenv_mod(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_hardcoded_credentials

Diagnose Hardcoded Credentials in Package Code

Description

Flags a secret that looks like an API key, access token, or private key sitting in a string literal in R/. R CMD check does not scan for leaked credentials, yet a token committed to a package is a security problem: once the package is published to CRAN the secret is public and must be revoked.

Usage

```
lab_hardcoded_credentials(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

path	Character. Path to package directory.
verbose	Logical. Print diagnostic messages.
parsed	Internal. Pre-parsed source cache; if NULL, files are read from path on demand.

Details

Only string literals in the parsed sources are examined, so the same text in a comment or a variable name never matches, and the check's own pattern strings never flag themselves.

Each format is anchored on a provider-specific prefix so ordinary code does not match. The recognised credentials, and the prefix each is keyed on, are:

- **Version control and registries**: GitHub tokens (ghp_, gho_, ghu_, ghs_, ghr_, and fine-grained github_pat_), GitLab tokens (glpat-), npm tokens (npm_), and PyPI upload tokens (pypi-AgEIcHlwaS5vcmc).
- **Cloud providers**: AWS access keys (AKIA, ASIA, AROA, AIDA, AGPA, ABIA, ACCA), Google API keys (AIza), Google OAuth client secrets (GOCSPX-), and DigitalOcean tokens (dop_v1_).
- **AI and ML providers**: OpenAI keys (sk-, sk-proj-), Anthropic keys (sk-ant-), and Hugging Face tokens (hf_).
- **Payments**: Stripe secret and restricted keys (sk_live_, sk_test_, rk_live_, rk_test_) and Square tokens (sq0atp-, sq0csp-, EAAA).

- **Messaging**: Slack tokens (xoxb-, xoxp-, ...) and incoming webhooks (hooks.slack.com/services/...), SendGrid keys (SG.), and Telegram bot tokens (<id>:AA...).
- **Platforms**: Shopify tokens (shpat_, shpss_, shpca_, shppa_), Databricks tokens (dapi), and Postman keys (PMAK-).
- **Generic**: JSON Web Tokens (eyJ...) and PEM private-key headers (-----BEGIN ... PRIVATE KEY-----).

The prefixes and lengths follow each provider's published token format and the community-maintained gitleaks secret-detection ruleset.

Value

checktor_check_result() with passed, issues, message.

Source

No formal rule. A token or key committed to a package is public the moment it reaches CRAN and must be revoked, which is why this sits at robustness tier. See vignette("check-sources", package = "checktor") for how every check maps to its source.

References

Provider token formats and the gitleaks ruleset: https://github.com/gitleaks/gitleaks

Examples

```
pkg <- example_diagnose_scenario("code_examples/credentials_bad.R",
                                 show_content = FALSE)
lab_hardcoded_credentials(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_home_writing *Diagnose Writes to the User's Home Directory*

Description

Flags a write whose destination resolves to ~ or $HOME.

Usage

```
lab_home_writing(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.
parsed	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing path afresh.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Repository Policy states that "Packages should not write in the user's home filespace ... nor anywhere else on the file system apart from the R session's temporary directory". See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/home_writing_bad.R",
                                 show_content = FALSE)
lab_home_writing(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_identifier_format *Diagnose Author Identifier Formatting*

Description

Validates ORCID and ROR identifiers carried in Authors@R person comment fields, mirroring CRAN's bad_ORCID_iDs and bad_ROR_IDs incoming checks. ORCID iDs are checked against their checksum, ROR IDs against their shape.

Usage

```
lab_identifier_format(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.
desc	Optional pre-parsed DESCRIPTION, as returned by base::read.dcf(). Defaults to reading it from path.

Details

An Authors@R that cannot be read has no identifiers to judge, so the check is reported as skipped and lab_authors() reports the field.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN incoming check run by `R CMD check --as-cran` NOTEs a malformed ORCID or ROR identifier in `Authors@R`. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/identifier_format_bad.txt",
                                 show_content = FALSE)
lab_identifier_format(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_installed_packages

Diagnose installed.packages() Usage

Description

Flags `installed.packages()`, which is slow and is discouraged by its own help page.

Usage

```
lab_installed_packages(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`parsed`	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Repository Policy does not let a package install other packages when it runs. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/installed_packages_bad.R",
                                 show_content = FALSE)
lab_installed_packages(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_internal_ns *Diagnose* ::: *in Package Code*

Description

Flags a `pkg:::fn()` call in `R/`. The triple colon reaches an object another package does not export, whose behaviour its author is free to change in routine maintenance, so a release elsewhere can break your package without warning.

Usage

```
lab_internal_ns(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to package directory.
`verbose`	Logical. Print diagnostic messages.
`parsed`	Internal. Pre-parsed source cache; if `NULL`, files are read from `path` on demand.

Details

A call into your own package is reported too, since a package almost never needs `:::` for its own objects: everything in the namespace is already visible to the rest of it.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

CRAN sends this back as "Using foo:::f instead of foo::f allows access to unexported objects. This is generally not recommended ... Please omit one colon." `R CMD check` reports it too, under dependencies in R code, so this check is the same finding without waiting for a full check. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_example_internal_ns()` for the same rule in examples.

Examples

```
pkg <- example_diagnose_scenario("code_examples/internal_ns_bad.R",
                                 show_content = FALSE)
lab_internal_ns(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_language_names *Diagnose Programming-Language Names in DESCRIPTION*

Description

Flags a bare programming-language or statistical-computing name – Python, Java, JavaScript, Rust, MATLAB, SAS, Stata and more – in Title or Description that CRAN asks to see single-quoted. This is the language counterpart to lab_software_names(): both are policy-tier quoting checks, kept separate because a language name and a package name are different kinds of thing. R itself is never flagged in either form. No authority names it, and a bare R and a quoted 'R' both clear CRAN, so checktor takes no position on which you write. Single-letter or common-word names (C, Go, Swift) are left out because they cannot be told from ordinary prose.

Usage

```
lab_language_names(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.
desc	Optional pre-parsed DESCRIPTION, as returned by base::read.dcf(). Defaults to reading it from path.

Details

Data, markup and query formats (JSON, HTML, YAML, SQL, Markdown, LaTeX and the like) and the languages CRAN packages write either way (C++, Fortran, Tcl) are not flagged here. A census of CRAN in September 2026 found that packages accepted at new-package review in the previous 18 months wrote them bare 62% of the time, against 28% for the languages this check covers, so a bare one is not a policy finding. lab_format_names() reports them when you ask.

A package can extend the list through Config/checktor/language_names in its own DESCRIPTION.

Each occurrence is judged on its own, as in lab_software_names(): a term inside a quoted longer name such as 'MATLAB Runtime' or 'AWS Python SDK' is quoted, and so is one in a web address, a <doi:...> or a double-quoted book title.

Value

checktor_check_result() with passed, issues, message.

Source

Writing R Extensions, under "The DESCRIPTION file", asks for single quotes around other software; checktor applies the same to programming-language and markup names. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_software_names()`, `lab_format_names()`.

Examples

```
pkg <- example_diagnose_scenario("description_examples/language_names_bad.txt",
                                 show_content = FALSE)
lab_language_names(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_library_in_pkg *Diagnose library() in Package Code*

Description

Flags `library()` / `require()` in package code, which alters the user's search path. Code destined for a parallel worker is exempt, since a daemon starts with an empty search path.

Usage

```
lab_library_in_pkg(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`parsed`	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "Package Dependencies", asks package code to reach its dependencies through `Imports` and `::` rather than attaching them with `library()`. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/library_in_pkg_bad.R",
                                 show_content = FALSE)
lab_library_in_pkg(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_license *Diagnose the License Field*

Description

Flags a License that tools::analyze_license() cannot standardize, and a + file LICENSE pointing at a file that does not exist. For MIT and the BSD licenses it also reads that file as R CMD check does: it must be the stub of YEAR and COPYRIGHT HOLDER fields (and ORGANIZATION for BSD 3-clause), so the full license text in its place is reported, as R's "License stub is invalid DCF" NOTE.

Usage

```
lab_license(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.
desc	Optional pre-parsed DESCRIPTION, as returned by base::read.dcf(). Defaults to reading it from path.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Repository Policy treats an invalid or unrecognised License field as a rejection. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/license_bad.txt",
                                 show_content = FALSE)
lab_license(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_license_file_unneeded

Diagnose an Unneeded LICENSE File Pointer

Description

Flags `+ file LICENSE` added to a standard license R knows, such as `GPL-3 + file LICENSE`, `LGPL-3 + file LICENSE` or `Apache License 2.0 + file LICENSE`. MIT and the BSD licenses are templates that need a `LICENSE` file naming the year and copyright holder, so they are left alone, as is a bare `file LICENSE`. Each alternative of a dual license such as `GPL-3 + file LICENSE | MIT + file LICENSE` is judged on its own.

Usage

```
lab_license_file_unneeded(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Details

A `LICENSE` that adds attribution requirements or other restrictions is the exception CRAN allows. Explain it in `cran-comments.md` and allow the finding with `Config/checktor/allow: license_file_unneeded`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook's LICENSE files recipe gives the reviewers' text: "We do not need \"+ file LICENSE\" and the file as these are part of R. This is only needed in case of attribution requirements or other possible restrictions. Hence please omit it." R agrees in two places. `R CMD check` NOTEs "License components with restrictions not permitted" for a license that takes no extension, such as `GPL (>= 2)` or Apache, and the incoming check NOTEs "License components with restrictions and base license permitting such" for one that does, such as `GPL-3`. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`lab_license()` for a license R cannot read; `checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/license_file_unneeded_bad.txt",
                                 show_content = FALSE)
lab_license_file_unneeded(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_license_year *Diagnose an Unfilled LICENSE Template*

Description

Flags a LICENSE file still carrying template placeholders such as <YEAR> or <COPYRIGHT HOLDER>.

Usage

```
lab_license_year(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Cookbook covers the file itself under LICENSE files. An unfilled template, with <YEAR> or <COPYRIGHT HOLDER> left in, leaves a placeholder, and no binding rule names it, which is why this sits at robustness tier. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/license_year_bad.LICENSE",
                                 show_content = FALSE)
lab_license_year(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_missing_examples *Diagnose Exported Functions Missing Examples*

Description

CRAN expects exported functions to carry a runnable `\examples{}` section. Walks `.Rd` files via `tools::parse_Rd()` and reports exported function topics that lack one. Data, class, methods, package-level, and re-export topics are skipped, and only topics whose name appears in NAMESPACE `export()` are considered (so internal helpers and S3 methods aren't required to have examples). A function that exists only for its side effect may be reported here even though it is fine, so use your judgement.

Usage

```
lab_missing_examples(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Value

`checktor_check_result()` with `passed`, `issues`, `missing`, `message`.

Source

The CRAN Cookbook covers examples under Structuring of Examples. Exported functions are expected to carry an `\examples{}` block, a convention rather than a rule, which is why this sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/missing_examples_bad.Rd",
                                 show_content = FALSE)
writeLines("export(undocumented_fn)", file.path(pkg, "NAMESPACE"))
lab_missing_examples(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_network_operations

Diagnose Unguarded Network Access

Description

Flags network access in code or examples that runs without a guard.

Usage

```
lab_network_operations(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to the package directory. Default: `".."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.

Details

An example is read as parsed R, so a function named in a comment, a string or an Rd % comment is not a call. A request counts as guarded when it sits in `\dontrun{}` or `\donttest{}`, or in the branch of an `if` whose condition confines it to a connection or a session: `curl::has_internet()`, `pingr::is_online()`, `capabilities("libcurl")` or `interactive()`, including roxygen's `@examplesIf` with any of them. Only a guard that encloses the call excuses it, and a guard kept in a variable counts only when it is assigned in the same function and on every path to the test. A request function handed to something that calls it, as in `lapply(urls, download.file)`, `purrr::map(reqs, httr2::req_perform)`, `do.call(httr::GET, args)` or `do.call("download.file", args)`, is a request too, and so is one called through parentheses, as in `(download.file)(u, f)`, or wrapped by `Vectorize()`, `memoise()` or an adverb such as `purrr::possibly()`, which return a function that calls it. Named anywhere else, as in `args(download.file)` or a mock that redefines it, it is not. Neither is a helper from a network package that only builds a request, such as `curl::form_file()`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Repository Policy states that "Packages which use Internet resources should fail gracefully with an informative message if the resource is not available". See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("network_examples/bad_network_example.Rd",
                                 show_content = FALSE)
lab_network_operations(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_news_file *Diagnose a Missing NEWS File*

Description

CRAN expects packages (especially on resubmission) to document user-facing changes in a `NEWS` file. Accepts `NEWS.md`, `NEWS`, or `NEWS.Rd` at the package root or under `inst/`.

Usage

```
lab_news_file(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

No formal rule. A `NEWS` file is expected but not required, a convention which is why this sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
invisible(file.remove(file.path(pkg, "NEWS.md")))
lab_news_file(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_option_changes *Diagnose Unrestored Option Changes*

Description

Flags a call to `options()`, `par()` or `setwd()` that changes the user's session state without restoring it.

Usage

```
lab_option_changes(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`parsed`	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Change of Options, Graphical Parameters and Working Directory, which asks that a function restore any `options()`, `par()` graphics parameters, or working directory it changes, using `on.exit()`. No clause in the Repository Policy or Writing R Extensions states it, yet it is among the most common reasons a package is sent back, because a function that quietly runs `options(digits = 3)` and returns has changed how the rest of the session behaves. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Exemptions

Two shapes leak nothing and are exempt. A setter that captures the old value and hands it back, as in `old <- options(...); invisible(old)`, honours the base R contract and lets the caller restore. And an option in the package's own namespace, written `options(<PackageName>.key = ...)`, is the package managing its own configuration rather than the user's, so a deliberate session preference like `options(mypkg.threshold = 5)` is fine. If you keep a user setting for the session, give it a namespaced name rather than a bare global one, and restore a genuinely temporary change with `on.exit(options(old))`.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/option_changes_bad.R",
                                 show_content = FALSE)
lab_option_changes(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_package_size *Diagnose Package Size*

Description

Estimates the size of the source package that would be sent to CRAN (files matched by `.Rbuildignore`, plus standard scratch directories like `.git`, `.Rproj.user`, are excluded). Warns at the 5 MB threshold.

Usage

```
lab_package_size(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Value

`checktor_check_result()` with `passed`, `issues`, `message`, and `size_mb`.

Source

The CRAN Repository Policy limits the size of the built tarball. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
# About 7 MB of data that gzip cannot shrink much
dir.create(file.path(pkg, "inst", "extdata"), recursive = TRUE)
writeBin(sin(seq_len(1e6) * 1.1), file.path(pkg, "inst", "extdata", "series.bin"))
lab_package_size(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_print_cat_usage *Diagnose Print/Cat Usage in Functions*

Description

Flags `print()` / `cat()` calls not guarded by an enclosing `if()`, `for()`, or `while()`. The check uses the ancestor axis, so guard detection is robust regardless of formatting. Calls inside S3 `print.*` and `format.*` methods are exempt, since `cat()` is the required idiom there (base R's own `print.default()` / `print.lm()` use it).

Usage

```
lab_print_cat_usage(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to package directory.
`verbose`	Logical. Print diagnostic messages.
`parsed`	Internal. Pre-parsed source cache; if `NULL`, files are read from `path` on demand.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Using print()/cat(). Diagnostic output belongs in `message()` or `warning()`, which a user can suppress, rather than in `cat()` or `print()`, which they cannot. Neither the Repository Policy nor Writing R Extensions states this, but reviewers ask for it consistently. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("code_examples/print_cat_bad.R",
                                 show_content = FALSE)
lab_print_cat_usage(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_rd_bibliography *Diagnose Citations Missing From the Bibliography*

Description

Reads the \bibcitet{}, \bibcitep{} and \bibshow{} macros R 4.6.0 added to .Rd files and reports what R CMD check would: a key that no bibliography holds, which R drops from the page, and a key cited in a file whose \bibshow{} never lists it. A REFERENCES file at the top level of the package is reported too, since R CMD check --as-cran notes it as a non-standard file.

Usage

```
lab_rd_bibliography(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to the package directory. Default: ".".
verbose	Logical. Print diagnostic output. Default: TRUE.

Details

A key is looked up as R looks it up: in the package's first REFERENCES.rds, REFERENCES.R or REFERENCES.bib, in the root and then inst/, and in R's own bibliography. A pkg::key is looked up in that package's bibliography when the package is in Depends, Imports or Suggests and installed here. Nothing is run to read the files: a key REFERENCES.R does not give as a string is not guessed at, and no key is reported missing from a bibliography that could not be read.

Value

checktor_check_result() with passed, issues, message. It is skipped when R's own bibliography, which R 4.6.0 added, is not there to look a key up in.

Source

Writing R Extensions, under "Bibliographic citations and references", describes the macros and says the bibliography is kept in inst/. R CMD check in R 4.6 warns "Could not find bibentries for the following keys" and notes "Bibentries cited but not shown in Rd file", and --as-cran notes a "Non-standard file/directory found at top level". See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), lab_rd_bibliography_files().

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/rd_bibliography_bad.Rd",
                                 show_content = FALSE)
lab_rd_bibliography(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_rd_bibliography_files

Diagnose a Bibliography R Cannot Read or Install

Description

For a package that cites with R 4.6.0's `\bibcitet{}`, `\bibcitep{}` or `\bibshow{}` macros, reports a `REFERENCES.bib` whose package, `bibtex`, is not in `Depends`, `Imports` or `Suggests`, and a `REFERENCES` file that is not installed because it is outside `inst/` or excluded by `.Rbuildignore`.

Usage

```
lab_rd_bibliography_files(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.

Details

R reads the bibliography when it builds the package, wherever it is, so the package's own pages are fine. But a `.bib` file is read with `bibtex::read.bib()`, so a build on a machine that installed only the declared dependencies, as r-universe and pak do, has no way to read it. And R installs only what is in `inst/`, while another package's `\bibcitet{pkg::key}` looks for the bibliography in the installed package.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions makes the bibliography available as `inst/REFERENCES.R` or `inst/REFERENCES.bib`, the latter "needs bibtex to be installed". No check enforces either, so this is `robustness` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, `lab_rd_bibliography()`.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/rd_bibliography_bad.Rd",
                                 show_content = FALSE)
# Give the page a REFERENCES.bib, without 'bibtex' in Suggests to read it
dir.create(file.path(pkg, "inst"))
writeLines("@book{smith2020, author = {Ann Smith}, title = {Smoothing}, year = {2020}}",
           file.path(pkg, "inst", "REFERENCES.bib"))
lab_rd_bibliography_files(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_readme_links *Diagnose Relative Links in the README*

Description

Relative links in `README.md`/`README.Rmd` render on GitHub but break on CRAN when the target is not included in the built tarball. This flags relative links whose target is missing on disk or excluded by `.Rbuildignore` (and therefore absent after `R CMD build`). Relative links to files that are included (e.g. `man/figures/logo.png`) are not flagged.

Usage

```
lab_readme_links(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Details

Only what a renderer turns into a link counts. Fenced code blocks, inline code spans and HTML comments are left out of the scan, so a setup chunk calling `knitr::opts_chunk[["set"]](...)` is read as the code it is rather than as a link to a file named after its arguments.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

No formal rule. A relative README link whose target is excluded from the tarball breaks on the package page, which is why this sits at `robustness` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
# A relative link to a file the package does not have
writeLines("See [the guide](docs/guide.md) for details.",
           file.path(pkg, "README.md"))
lab_readme_links(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

`lab_references` *Diagnose Reference Formatting in DESCRIPTION*

Description

Flags the links in the `Description` that CRAN's incoming check NOTEs, applying its own rules to the same wrapped lines R reads:

Usage

```
lab_references(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Details

- a URL not enclosed in angle brackets, which should read `<https://...>`;
- a DOI not written as `<doi:prefix/suffix>`, such as a `https://doi.org/` link, a bare `doi:`, `<doi` with no colon, or `<10.xxxx/...>`;
- a publisher link that embeds a DOI, such as `<https://onlinelibrary.wiley.com/doi/10.1002/...>`, reported only when no DOI is malformed, as R does;
- an arXiv id or link, such as `<arXiv:1509.03700>` or `<https://arxiv.org/abs/...>`, which should be the e-print's arXiv DOI, `<doi:10.48550/arXiv.1509.03700>`.

It also flags a reference with no closing `>`, which CRAN's page does not render as a link. Each rule is one finding, quoting every reference that breaks it. A space after the colon, as in `<doi: 10.1000/xyz>`, is left alone: R does not NOTE it and CRAN's page links it all the same.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN incoming check run by `R CMD check --as-cran` NOTEs each of the four forms, with "Please enclose URLs in angle brackets (<...>).", "Please write DOIs as `doi:prefix/suffix`.", "Please use permanent DOI markup for linking to publications as in `doi:prefix/suffix`." and "Please refer to arXiv e-prints via their arXiv DOI `doi:10.48550/arXiv.YYMM.NNNNN`.". The submission checklist says "arXiv preprints should be referred to via their arXiv DOI", and the Cookbook's References recipe asks for "angle brackets for auto-linking". `devtools::check()` turns the incoming check off, so these usually surface first on win-builder or at CRAN. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/references_bad.txt",
                                 show_content = FALSE)
lab_references(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_roxygen_usage *Diagnose Stale Generated Documentation*

Description

Flags a package whose `NAMESPACE` and `man/` no longer match the roxygen comments they were generated from, i.e. you edited roxygen and forgot to run `devtools::document()`.

Usage

```
lab_roxygen_usage(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Details

Two signals, both deliberately clock-free:

- A name tagged `@export` in a roxygen block that does not appear in `NAMESPACE`. This is the real cost of a forgotten `document()` call: the function is not exported, so users cannot call it, and `R CMD check` says nothing because a package is free to export whatever it likes.
- An `.Rd` file whose roxygen2 backlink names a source file that no longer exists, which is the orphan left behind when a source file is renamed or deleted without re-documenting.

File modification times are deliberately not used. `git` does not preserve them, so on a fresh clone every file carries roughly the checkout time in arbitrary order, and an mtime comparison would pass or fail at random in CI.

The check is skipped entirely unless `NAMESPACE` carries the roxygen2 "do not edit by hand" banner, so a hand-managed package is never flagged.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers the roxygen2 side of this under Repeated Rejections of Issues in Manuals, where documentation is regenerated rather than hand-edited. A function tagged `@export` that never reached `NAMESPACE` is not actually exported, and no binding rule names it, which is why this sits at `robustness` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("code_examples/roxygen_usage_bad.R",
                                 show_content = FALSE)
# NAMESPACE as roxygen2 wrote it before rescale01() was tagged @export
writeLines("# Generated by roxygen2: do not edit by hand",
           file.path(pkg, "NAMESPACE"))
lab_roxygen_usage(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_seed_setting *Diagnose Hardcoded Seed Setting*

Description

Flags `set.seed(<numeric>)` calls. Multi-line forms are handled because the check matches the call AST node, not raw text.

Usage

```
lab_seed_setting(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to package directory.
`verbose`	Logical. Print diagnostic messages.
`parsed`	Internal. Pre-parsed source cache; if `NULL`, files are read from `path` on demand.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Repository Policy asks a package not to modify the user's workspace, and set.seed() writes .Random.seed there, changing the random-number stream for the rest of the session. See vignette("check-sources", package = "checktor") for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("code_examples/seed_setting_bad.R",
                                 show_content = FALSE)
lab_seed_setting(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_software_install *Diagnose Package Installation From Package Code*

Description

Flags install.packages() and friends. A package may not install software on the user's machine.

Usage

```
lab_software_install(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

path	Character. Path to the package directory. Default: ". ".
verbose	Logical. Print diagnostic output. Default: TRUE.
parsed	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing path afresh.

Value

checktor_check_result() with passed, issues, message.

Source

The CRAN Repository Policy does not let a package download and install external software when it loads or runs. See vignette("check-sources", package = "checktor") for how every check maps to its source.

See Also

checktor(), which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/software_install_bad.R",
                                 show_content = FALSE)
lab_software_install(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

`lab_software_names` *Diagnose Unquoted Software Names in DESCRIPTION*

Description

Flags a package or external-software name in `Title`/`Description` that is not in single quotes, as Writing R Extensions requires.

Usage

```
lab_software_names(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `"."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Details

Each occurrence is judged on its own, so one quoted mention does not excuse a bare one elsewhere. A name is not bare inside a single-quoted span such as `'shiny.semantic'`, a `<https://...>` or `<doi:...>`, or a function call such as `purrr::map()`, which are the spans CRAN's own incoming spell check skips. Nor is it bare in a plain web address, or in a double-quoted quotation such as the title of a book. A name alone in double quotes, in either field, is reported by `lab_description_quoted_quotes()` instead.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "The DESCRIPTION file", asks you to "Refer to other packages and external software in single quotes". See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/software_names_bad.txt",
                                 show_content = FALSE)
lab_software_names(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_spelling *Diagnose Possibly Misspelled Words in DESCRIPTION*

Description

Spell-checks the `Title` and `Description` fields with `utils::aspell()`, mirroring the aspell pass in CRAN's incoming check. It reports only words that are not already accepted somewhere: a package `.aspell/` dictionary, `inst/WORDLIST`, or a `Config/checktor/acronyms`, `software_names`, `language_names` or `format_names` field.

Usage

```
lab_spelling(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Present for signature parity with the other DESCRIPTION checks; spelling reads the `DESCRIPTION` file directly and ignores it.

Details

The check needs a spell-check backend (`aspell` or `hunspell`) on the system. Without one it is reported as skipped, the same way CRAN's incoming check skips spelling when no backend is present, so a run on one machine may find words a run on another does not. It is therefore an `opinion`-tier check. When it does fire, `prescribe()` hands back a ready-to-paste `.aspell/` snippet with the flagged words filled in.

Value

`checktor_check_result()` with `passed`, `issues`, `message`; the `issues` are the possibly-misspelled words.

Source

The CRAN incoming check runs `aspell` over the `Title` and `Description`; it needs a spell-check backend and is noisy, so checktor keeps it at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

checktor(), prescribe().

Examples

```
# Needs aspell or hunspell; without one the check is skipped and finds nothing
pkg <- example_diagnose_scenario("description_examples/spelling_bad.txt",
                                 show_content = FALSE)
lab_spelling(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_suggested_in_examples

Diagnose Suggested Packages Used in Examples Without a Guard

Description

Under CRAN's noSuggests check a package must work without its Suggested packages installed. This flags an example that needs a Suggested package, through pkg::, library() or require(), in code that runs without a guard naming that package: an enclosing if (requireNamespace("pkg", quietly = TRUE)) or if (require("pkg")), including roxygen's @examplesIf with one of them. rlang::is_installed("pkg") counts too, but it needs rlang, so it is a use of rlang when rlang is only suggested.

Usage

```
lab_suggested_in_examples(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to package directory
verbose	Logical. Print diagnostic messages

Details

The example is read as parsed R, so a package named in a comment or a string is not a use, and a guard in a comment, a guard for another package, or one that does not enclose the use excuses nothing. Usage inside \dontrun{} is not flagged, since it never runs. Usage inside \donttest{} is, since R CMD check --as-cran runs it. A condition that is false under R CMD check keeps the use out of the check as surely as \dontrun{} does, so interactive(), identical(Sys.getenv("IN_PKGDOWN"), "true"), nzchar(Sys.getenv("IN_PKGDOWN")) and a test that NOT_CRAN is "true" excuse it too. A bare as.logical(Sys.getenv("NOT_CRAN")) does not: it is NA there, and if (NA) stops the example with an error. R's base packages, such as parallel and tools, and its recommended packages, such as MASS, Matrix and survival, ship with R and are never flagged.

Value

checktor_check_result() with passed, issues, message.

Source

Writing R Extensions, under "Suggested packages", asks that a package from `Suggests` used in an example be guarded so the example still runs without it. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/suggested_in_examples_bad.Rd",
                                 show_content = FALSE)
cat("Suggests: somesuggest\n",
    file = file.path(pkg, "DESCRIPTION"), append = TRUE)
lab_suggested_in_examples(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_system_calls *Diagnose System Calls*

Description

Flags `system()` / `system2()` / `shell()`, which need review for portability and for shell-injection risk.

Usage

```
lab_system_calls(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`parsed`	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

No flat rule. A raw `system()` or `system2()` call needs review for portability rather than being an automatic violation, which is why this sits at `robustness` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/system_calls_bad.R",
                                 show_content = FALSE)
lab_system_calls(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_sys_setenv *Diagnose Unrestored Environment Variables*

Description

Flags `Sys.setenv()` with no matching cleanup in the same function.

Usage

```
lab_sys_setenv(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`parsed`	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

No clause names environment variables, but they are session state exactly as `options()` are, so the same restore-on-exit requirement applies. The CRAN Cookbook states it for options under Change of Options, Graphical Parameters and Working Directory. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/sys_setenv_bad.R",
                                 show_content = FALSE)
lab_sys_setenv(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_temp_cleanup *Diagnose Missing Temp-File Cleanup*

Description

Flags a temporary file created without a matching cleanup.

Usage

```
lab_temp_cleanup(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

path	Character. Path to the package directory. Default: `"."`.
verbose	Logical. Print diagnostic output. Default: `TRUE`.
parsed	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Leaving Files in the Temporary Directory. `tempdir()` is removed at session end, so an un-`unlink()`ed tempfile breaks no rule, and tidiness rather than policy is why this sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("temp_examples/bad_temp_usage.R",
                                 show_content = FALSE)
lab_temp_cleanup(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_tf_usage *Diagnose* T/F *Usage in R Code*

Description

Flags bare T / F symbols that should be TRUE / FALSE. Operates on the parsed syntax tree, so T inside a string or a comment is not reported, which a plain text search could not tell apart. Named-argument names (f(T = 1)) and $T / @T extractions are excluded.

Usage

```
lab_tf_usage(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

path	Character. Path to package directory.
verbose	Logical. Print diagnostic messages.
parsed	Internal. Pre-parsed source cache; if NULL, files are read from path on demand.

Value

checktor_check_result() with passed, issues, message.

Source

No binding rule forbids T and F, though the CRAN Cookbook keeps a recipe for it under T/F Instead of TRUE/FALSE. They are ordinary variables (see ?logical) that R sets to TRUE and FALSE at startup but that any code can rebind, so a function reading T after something has run T <- 0 gets the wrong answer. A real risk that no rule makes citable is why this sits at robustness tier rather than policy. See vignette("check-sources", package = "checktor") for how every check maps to its source.

Examples

```
# show_content defaults to TRUE, so the offending file prints first
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R")
lab_tf_usage(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_title_case — *Diagnose Title Case in DESCRIPTION*

Description

Flags a `Title` that is not in title case, delegating to `tools::toTitleCase()`, which restores single-quoted spans so `'shiny'` keeps its own capitalisation.

Usage

```
lab_title_case(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "The DESCRIPTION file", says the `Title` "should use title case"; the `-as-cran` incoming check flags one that does not. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/title_case_bad.txt",
                                 show_content = FALSE)
lab_title_case(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_title_length *Diagnose Title Length*

Description

Flags a `Title` longer than 65 characters.

Usage

```
lab_title_length(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions §1.1.1 notes that some package listings may truncate the title to 65 characters. That is a display width rather than a limit, so a title of exactly 65 characters still shows in full and only a longer one loses its tail. Nothing rejects a long title, which is why this sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/title_length_bad.txt",
                                 show_content = FALSE)
lab_title_length(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_title_package_name

Diagnose a Title That Repeats the Package Name

Description

Flags a `Title` that is just the package name, or that opens with the name followed by a colon, as in `toypkg: Fit Simple Models`. Package listings already show the name beside the `Title`, so it reads twice.

Usage

```
lab_title_package_name(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Details

R's incoming check also NOTEs a `Title` that opens with the name followed by a space, but that form is left alone here: when the name is an ordinary word, as in survival's "Survival Analysis", CRAN accepts it routinely, and over a hundred packages on CRAN carry one. R drops the NOTE for an update whose `Title` is unchanged since the version on CRAN, which checktor cannot see offline, so an older package may pass CRAN with a finding here.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "The DESCRIPTION file", says of the `Title`: "Do not repeat the package name: it is often used prefixed by the name." The incoming check NOTEs "The Title field is just the package name: provide a real title." and "The Title field starts with the package name.". See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/title_package_name_bad.txt",
                                 show_content = FALSE)
lab_title_package_name(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

---

lab_title_redundant_phrases

Diagnose Redundant Phrases in Title

Description

Flags a `Title` carrying a phrase CRAN asks you to drop, such as "for R".

Usage

```
lab_title_redundant_phrases(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

No formal rule. Phrases like "R package to" are redundant in a `Title`, a convention which is why this sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/title_redundant_phrases_bad.txt",
                                 show_content = FALSE)
lab_title_redundant_phrases(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_title_starts_with_article

Diagnose Title Starting With an Article

Description

Flags a `Title` beginning with `A`, `An`, or `The`.

Usage

```
lab_title_starts_with_article(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

No rule. This was a mis-transplant of a real CRAN rule and is kept callable but off by default. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/title_starts_with_article_bad.txt",
                                 show_content = FALSE)
lab_title_starts_with_article(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_unexported_example_ns

Diagnose Bare Calls to Unexported Functions in Examples

Description

Flags an \examples{} block that calls its own topic bare when that topic is not exported. Examples run with only the package's exports attached, so the call fails.

Usage

```
lab_unexported_example_ns(path = ".", verbose = TRUE)
```

Arguments

path	Character. Path to package directory
verbose	Logical. Print diagnostic messages

Details

R CMD check does catch this, but only by RUNNING the examples, which is late and slow, and it is skipped entirely when examples are wrapped in \dontrun{} or when you check with --no-examples. This finds it statically in a second.

Value

checktor_check_result() with passed, issues, message.

Source

No formal rule. An example that reaches for an unexported object will error when it runs, which is why this sits at robustness tier. See vignette("check-sources", package = "checktor") for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/unexported_example_ns_bad.Rd",
                                 show_content = FALSE)
# The package exports something, but not internal_values()
writeLines("export(public_values)", file.path(pkg, "NAMESPACE"))
lab_unexported_example_ns(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_urls — *Diagnose URL Issues in Package Files*

Description

Flags `http://` URLs (which should almost always be `https://`) and known URL shortener domains, across DESCRIPTION, README, vignettes and the `.Rd` files.

Usage

```
lab_urls(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Details

This is a fast, **offline** pre-flight. `R CMD check --as-cran` does fetch every URL and report status codes and redirect targets, but only with a network and only under `--as-cran`, which is the slow end of the loop. This catches the two problems you can find without leaving the room, before you spend ten minutes on a full check.

Literal spans are skipped, so documenting the string `http://` inside `\verb{}`, `\code{}` or a fenced markdown block is not mistaken for linking to it.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

No formal rule. Preferring `https://` is good advice, but CRAN's NOTE is about broken URLs rather than the scheme, which is why this sits at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("description_examples/bad_description.txt",
                                 show_content = FALSE)
lab_urls(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_url_liveness *Diagnose Broken and Redirecting URLs*

Description

Fetches every URL in the package, across DESCRIPTION, `.Rd` files and vignettes, and reports the ones that fail: 404s, other error statuses, and redirects that ought to point at their final target. This is what `R CMD check --as-cran` does, through the same base R machinery (`tools::check_package_urls()`), so you can see those findings without a full `--as-cran` pass and without depending on the `urlchecker` package.

Usage

```
lab_url_liveness(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Details

It runs by default when you are working at the console, since that is when a broken link is worth knowing about and a pause for the network is fine. It stays off in scripts, in continuous integration and under `R CMD check`, where a slow or unreachable network would make results depend on the machine rather than the package. Set the option either way to decide for yourself:

```
options(checktor.url_check = TRUE)   # always check
options(checktor.url_check = FALSE)  # never check
```

Without a network nothing can be fetched, so the check is reported as skipped rather than as passing. For the offline half, which flags `http://` links and URL shorteners without leaving the room, see `lab_urls()`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN incoming check run by `R CMD check --as-cran` fetches URLs and NOTEs 404s and redirects. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
# Needs a network: it asks every URL the package lists whether it answers,
# so it is not run here. The scenario lists one page that exists and one that
# does not, and the missing one is reported with its 404. Outside an
# interactive session the check is skipped unless
# options(checktor.url_check = TRUE) turns it on.
## Not run:
pkg <- example_diagnose_scenario("general_examples/url_liveness_bad.txt",
                                 show_content = FALSE)
lab_url_liveness(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)

## End(Not run)
```

---

lab_value_tags *Diagnose Missing Value Tags in Documentation*

Description

Walks `.Rd` files via `tools::parse_Rd()` and reports topics that are missing a `\value{}` section. Data, class, methods, package-level, and re-export topics are skipped (they don't need `\value{}`).

Usage

```
lab_value_tags(path = ".", verbose = TRUE)
```

Arguments

`path`	Character. Path to package directory
`verbose`	Logical. Print diagnostic messages

Value

`checktor_check_result()` with `passed`, `issues`, `missing`, `message`.

Source

Writing R Extensions, under "Documenting functions", describes `\value{}`, and the CRAN Cookbook keeps the recipe reviewers cite under Missing value-tags in .Rd-files, but `R CMD check` does not require it, so checktor keeps this at `opinion` tier. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

Examples

```
pkg <- example_diagnose_scenario("documentation_examples/missing_value_tag.Rd",
                                 show_content = FALSE)
lab_value_tags(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_version_format *Diagnose the DESCRIPTION Version Field*

Description

Flags a `Version` with a leading-zero component or a suspiciously large one, mirroring CRAN's `version_with_leading_zeroes` and `version_with_large_components` incoming checks. A calendar-year component (e.g. a dated `2026.01` version) is exempt.

Usage

```
lab_version_format(path = ".", verbose = TRUE, desc = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`desc`	Optional pre-parsed `DESCRIPTION`, as returned by `base::read.dcf()`. Defaults to reading it from `path`.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

Writing R Extensions, under "The DESCRIPTION file", says a `Version` is "a sequence of at least two ... non-negative integers"; the incoming check flags a leading zero or an implausible value. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("description_examples/version_format_bad.txt",
                                 show_content = FALSE)
lab_version_format(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

lab_warn_option *Diagnose Changes to options(warn=)*

Description

Flags a change to `options(warn = )` that is not restored.

Usage

```
lab_warn_option(path = ".", verbose = TRUE, parsed = NULL)
```

Arguments

`path`	Character. Path to the package directory. Default: `".."`.
`verbose`	Logical. Print diagnostic output. Default: `TRUE`.
`parsed`	Optional pre-parsed source, as returned internally by the orchestrator. Defaults to parsing `path` afresh.

Value

`checktor_check_result()` with `passed`, `issues`, `message`.

Source

The CRAN Cookbook covers this under Setting options(warn = -1). Suppressing warnings for the rest of the session hides them from everything that runs afterwards, so restore the previous value via `on.exit()`. See `vignette("check-sources", package = "checktor")` for how every check maps to its source.

See Also

`checktor()`, which runs this and every other check.

Examples

```
pkg <- example_diagnose_scenario("code_examples/option_changes_bad.R",
                                 show_content = FALSE)
lab_warn_option(pkg, verbose = FALSE)$issues
unlink(pkg, recursive = TRUE)
```

not_under_fn_with_call_xpath

XPath Predicate: Not Guarded by a Sibling Call

Description

Returns an XPath predicate that restricts matches to nodes whose innermost enclosing function body does *not* also contain a call to any of `funs`. This is how a check enforces a guard, for instance that an `options()` call is paired with an `on.exit()` in the same function.

Usage

```
not_under_fn_with_call_xpath(funs)
```

Arguments

`funs`	Character vector of guard function names (e.g. `"on.exit"`).

Details

The predicate anchors on `ancestor::expr[FUNCTION][1]`, the nearest function definition, and searches its whole subtree. Anchoring on the innermost function keeps it correct where the guard belongs to an inner function rather than an outer one, and covers a call sitting in a default argument as well as one in the body.

Value

A single character string: an XPath predicate to splice into a query after a node test.

See Also

`xpath_lints()`.

Examples

```
predicate <- not_under_fn_with_call_xpath(c("on.exit", "local_options"))
paste0("//SYMBOL_FUNCTION_CALL[text() = 'options'][", predicate, "]")
```

predicates *Status predicates for checktor results*

Description

These ask the question a verdict asks: did a check fail? `passed()` means "did not fail", so a check that did not run, such as `url_liveness` away from the console, is `TRUE` there. A skipped check cannot fail a verdict, and `is_healthy()`, `failed_checks()` and `n_failed_checks()` treat it the same way.

Usage

```
passed(x, ...)

## S3 method for class 'checktor_check_result'
passed(x, ...)

## S3 method for class 'checktor_category_result'
passed(x, ...)

## S3 method for class 'checktor_results'
passed(x, ...)

is_healthy(x, ...)

## S3 method for class 'checktor_check_result'
is_healthy(x, ...)

## S3 method for class 'checktor_category_result'
is_healthy(x, ...)

## S3 method for class 'checktor_results'
is_healthy(x, ...)

n_issues(x, ...)

## S3 method for class 'checktor_check_result'
n_issues(x, ...)

## S3 method for class 'checktor_category_result'
n_issues(x, ...)

## S3 method for class 'checktor_results'
n_issues(x, ...)

n_failed_checks(x, ...)
```

```
## S3 method for class 'checktor_category_result'
n_failed_checks(x, ...)

## S3 method for class 'checktor_results'
n_failed_checks(x, ...)

failed_checks(x, ...)

## S3 method for class 'checktor_category_result'
failed_checks(x, ...)

## S3 method for class 'checktor_results'
failed_checks(x, ...)
```

Arguments

x	A `checktor_results`, `checktor_category_result`, or `checktor_check_result` object.
...	Unused.

Details

tidy() answers a different question. It gives each check one state, so the same skipped check has `passed = FALSE` and `skipped = TRUE` there. To tell a pass from a skip, use `tidy()` or summary().

Value

`passed()`: logical, `TRUE` for a check that did not fail, a skipped one included. It is a single value for a check, a named logical by check for a category, and a named logical by category for results. `is_healthy()`: a single logical. `n_issues()` / `n_failed_checks()`: integer counts. `failed_checks()`: character vector of failing check names (qualified `"category.check"` at the results level).

On a `checktor_results`, `is_healthy()`, `n_issues()` and `n_failed_checks()` count only the tiers named in `checktor()`'s `severity` (they read `metadata$total_issues` and `metadata$failed_checks`), so they agree with `checkup()`. `failed_checks()`, and every predicate on a category or a check, count every tier, so an `opinion` finding can appear in `failed_checks()` and in `nrow(issues(x))` without changing `n_issues()`.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)
is_healthy(results)
failed_checks(results)
```

prescribe — *Treatment Recommendations*

Description

Prints specific treatment recommendations for the checks that failed in a `checktor()` run, whatever their severity tier, so an advisory finding outside the verdict still gets its remedy.

Usage

```
prescribe(results)
```

Arguments

`results`	A `checktor_results` object.

Value

Invisibly returns `NULL`. Called for the side effect of printing recommendations.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)
prescribe(results)
```

print.checktor_category_result — *Print Method for checktor_category_result Objects*

Description

Print Method for checktor_category_result Objects

Usage

```
## S3 method for class 'checktor_category_result'
print(x, ...)
```

Arguments

`x`	A checktor_category_result object
`...`	Additional arguments (unused)

Value

Returns x invisibly

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
print(diagnose_code_issues(pkg, verbose = FALSE))
```

---

print.checktor_check_result

Print Method for checktor_check_result Objects

Description

Print Method for checktor_check_result Objects

Usage

```
## S3 method for class 'checktor_check_result'
print(x, ...)
```

Arguments

x	A checktor_check_result object
...	Additional arguments (unused)

Value

Returns x invisibly

Examples

```
res <- checktor_check_result(FALSE, c("foo.R:3", "bar.R:9"), "T/F usage")
print(res)
```

print.checktor_results

Print Method for checktor_results Objects

Description

Provides a clean, formatted summary of diagnostic results from `checktor()`.

Usage

```
## S3 method for class 'checktor_results'
print(x, ...)
```

Arguments

`x`	A `checktor_results` object from `checktor()`
`...`	Additional arguments passed to print methods (currently unused)

Value

Returns `x` invisibly. Called primarily for its side effect of printing a formatted summary to the console.

See Also

`checktor()` to generate results, `health_report()` for detailed reports

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)
print(results)
```

read_r_xml

Parse a Package's R Sources into Queryable XML

Description

Parses every `R/*.R` file under `path` with `parse(keep.source = TRUE)` and converts each file's parse data to an `xml2` document via `xmlparsedata::xml_parse_data()`. This is the entry point for an AST-based check: hand the result to `xpath_lints()` or one of the other helpers. A syntax error in a file is caught and recorded in that file's `error` slot rather than crashing the run.

Usage

```
read_r_xml(path)
```

Arguments

path	Character. Path to the R package directory.

Value

A named list with one entry per R/*.R file, each a list of file (the path), xml (an xml2 document, or NULL if the file did not parse), and error (the simpleError, or NULL).

See Also

xpath_lints(), register_check(), and the *Writing Your Own Checks* vignette.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
parsed <- read_r_xml(pkg)
names(parsed)
```

---

registered_checks *List Registered Checks*

Description

Reports the checks currently added with register_check().

Usage

```
registered_checks()
```

Value

A data frame with columns check, category, and severity, one row per registered check (zero rows if none are registered).

See Also

register_check(), unregister_check()

Examples

```
registered_checks()
```

register_check *Register a Custom Check with* checktor()

Description

Adds a check of your own to every subsequent checktor() run without editing checktor's source. It joins the panel its category already runs, appears in issues(), tidy() and the printed report, and counts toward the verdict at the severity tier you declare.

Usage

```
register_check(
  name,
  fn,
  category = CHECK_CATEGORIES,
  severity = c("robustness", "policy", "opinion")
)
```

Arguments

name	Character. The check's key, used in results and reports. Must not clash with a built-in check name. Naming the function lab_<name>() keeps it consistent with the built-in checks, where the two always match.
fn	A function returning a checktor_check_result(), the same shape as any lab_* check. It is called as fn(path, verbose). If it also declares a parsed argument (for code and policy checks) or a desc argument (for description checks), checktor forwards its shared parse cache so the check does not re-read the sources. A check that cannot run where it is returns a skipped result, as checktor_check_result() describes, so it is reported as not run rather than as passing.
category	Character. Which category the check joins: one of "code", "description", "documentation", "general", "policy".
severity	Character. The tier the check reports at: one of "robustness" (default), "policy", or "opinion". This decides whether a finding counts against a clean bill of health under checktor()'s severity argument.

Details

A registered check joins one of the five built-in categories. There is no way to add a category of your own, so pick the panel your check belongs to.

Value

Invisibly, name.

See Also

unregister_check(), registered_checks(), checktor()

Examples

```
# A house rule: flag any call to a banned helper.
lab_no_banned <- function(path, verbose = TRUE, parsed = NULL) {
  if (is.null(parsed)) parsed <- read_r_xml(path)
  issues <- undesirable_function_check(parsed, "banned_helper")
  checktor_check_result(length(issues) == 0L, issues, "no banned_helper()")
}
register_check("no_banned", lab_no_banned,
               category = "code", severity = "policy")

registered_checks()
unregister_check("no_banned")
```

---

show_example_files *Show Available Example Files*

Description

Lists all available example files in the `inst/diagnose/` directory that can be used with `example_diagnose_scenario()`.

Usage

```
show_example_files(category = "all", pattern = NULL)
```

Arguments

`category`	Character. Optional category filter. One of "code", "description", "documentation", "general", "network", "temp", or "all". Default: "all".
`pattern`	Character. Optional regex pattern to filter filenames. Default: `NULL` (no filtering).

Value

Character vector of relative paths to example files that can be used with `example_diagnose_scenario()`.

See Also

`example_diagnose_scenario()` to create test scenarios with these files

Examples

```
# List all available examples
show_example_files()

# List only code examples
show_example_files("code")

# List files matching a pattern
show_example_files(pattern = "bad")

# Use with example_diagnose_scenario
examples <- show_example_files("code")
pkg_path <- example_diagnose_scenario(examples[1], show_content = FALSE)
unlink(pkg_path, recursive = TRUE)
```

---

summary.checktor_category_result

Per-category summary of checktor results

Description

Per-category summary of checktor results

Usage

```
## S3 method for class 'checktor_category_result'
summary(object, ...)

## S3 method for class 'checktor_results'
summary(object, ...)
```

Arguments

`object`	A `checktor_results` or `checktor_category_result` object.
`...`	Unused.

Value

For results: a 5-row `data.frame` (`category, checks, passed, failed, skipped, issues`). For a category: a 1-row `data.frame` (`checks, passed, failed, skipped, issues`). Each check is counted once, so `passed`, `failed` and `skipped` add up to `checks`, and a check that did not run is `skipped`, never `passed`.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)
summary(results)
```

tidy.checktor_results *Tidy a checktor result into a per-check data frame*

Description

Tidy a checktor result into a per-check data frame

Usage

```
## S3 method for class 'checktor_results'
tidy(x, ...)

## S3 method for class 'checktor_category_result'
tidy(x, ...)

## S3 method for class 'checktor_results'
as.data.frame(x, ...)

## S3 method for class 'checktor_category_result'
as.data.frame(x, ...)
```

Arguments

x	A `checktor_results` or `checktor_category_result` object.
...	Unused.

Value

A `data.frame` with one row per check: `category` (results level only), `check`, `severity`, `passed`, `skipped`, `n_issues`, `message`. `passed` is `TRUE` for a check that ran and found nothing. `skipped` marks a check that did not run, such as the URL fetch away from the console, and such a check is never `passed`. A check that failed is neither, so `!passed & !skipped` picks out the failures, and the three counts match summary(). `passed()` asks only whether a check failed, so it is `TRUE` for a skipped check that is `FALSE` here.

Examples

```
pkg <- example_diagnose_scenario("code_examples/tf_usage_bad.R",
                                 show_content = FALSE)
results <- checktor(pkg, verbose = FALSE, progress = FALSE)
tidy(results)
```

undesirable_function_check *Flag Every Call to a Named Function*

Description

The "flag any call to function X" pattern, checktor's analogue of `lintr::undesirable_function_linter()`. Member-access calls (`obj$fn(...)`, `obj@fn(...)`) are excluded, so only a genuine call to the bare function matches.

Usage

```
undesirable_function_check(parsed, funs, label = TRUE)
```

Arguments

`parsed`	A parsed-sources list from `read_r_xml()`.
`funs`	Character vector of function names to flag.
`label`	Logical. If `TRUE` (default), each hit is suffixed with the matched function name in parentheses.

Value

A character vector of `"basename:line"` strings.

See Also

`xpath_lints()`, `not_under_fn_with_call_xpath()`.

Examples

```
pkg <- example_diagnose_scenario("code_examples/browser_calls_bad.R",
                                 show_content = FALSE)
parsed <- read_r_xml(pkg)
undesirable_function_check(parsed, c("browser", "install.packages"))
```

unregister_check *Remove Registered Checks*

Description

Drops checks added with `register_check()`. With no argument, clears the whole registry.

Usage

```
unregister_check(name = NULL)
```

Arguments

name	Character vector of check names to remove, or NULL (the default) to remove every registered check.

Value

Invisibly, NULL.

See Also

register_check(), registered_checks()

Examples

```
register_check("tmp", function(path, verbose = TRUE) {
  checktor_check_result(TRUE, character(0), "noop")
})
unregister_check("tmp")
registered_checks()
```

---

xpath_lints *Collect XPath Matches as* file:line *Strings*

Description

Runs an XPath query against every parsed file from read_r_xml() and returns a "basename:line" string for each matching node, ready to use as a check's issues.

Usage

```
xpath_lints(parsed, xpath, label = NULL)
```

Arguments

parsed	A parsed-sources list from read_r_xml().
xpath	Character. An XPath 1.0 query, typically anchored on a SYMBOL_FUNCTION_CALL node.
label	Optional character. Appended in parentheses after each hit.

Value

A character vector of "basename:line" strings, empty if nothing matched.

See Also

read_r_xml(), xpath_per_file(), undesirable_function_check().

Examples

```
pkg <- example_diagnose_scenario("code_examples/seed_setting_bad.R",
                                 show_content = FALSE)
parsed <- read_r_xml(pkg)
xpath_lints(parsed, "//SYMBOL_FUNCTION_CALL[text() = 'set.seed']")
```

---

xpath_per_file *Summarise XPath Matches per File*

Description

A per-file variant of `xpath_lints()` for when you need to control the issue string. Runs `xpath` against each parsed file and calls `summarise(file, nodes)` on each non-empty match set, collecting the strings it returns.

Usage

```
xpath_per_file(parsed, xpath, summarise)
```

Arguments

`parsed`	A parsed-sources list from `read_r_xml()`.
`xpath`	Character. An XPath 1.0 query.
`summarise`	A function of `(file, nodes)` returning a character vector of issue strings, where `nodes` is an `xml2` nodeset.

Value

A character vector of the issue strings `summarise` produced.

See Also

`xpath_lints()`.

Examples

```
pkg <- example_diagnose_scenario("code_examples/seed_setting_bad.R",
                                 show_content = FALSE)
parsed <- read_r_xml(pkg)
xpath_per_file(parsed, "//SYMBOL_FUNCTION_CALL[text() = 'set.seed']",
               function(file, nodes) {
                 paste0(basename(file), ":", xml2::xml_attr(nodes, "line1"))
               })
```

Index

as.data.frame.checktor_category_result (tidy.checktor_results), 104
as.data.frame.checktor_results (tidy.checktor_results), 104

base::read.dcf(), *14*, *24*, *25*, *30*, *32–39*, *42*, *51*, *55*, *58*, *60*, *61*, *72*, *76*, *83–87*, *92*

checktor, 4
checktor(), *9*, *10*, *12–17*, *21*, *22*, *24–26*, *29–32*, *34–39*, *41–46*, *48–53*, *55–57*, *59–62*, *64*, *66*, *69*, *70*, *73*, *75*, *76*, *78–81*, *83–87*, *92*, *93*, *96*, *97*, *99*, *101*, *102*
checktor_category_result, 6
checktor_check_result, *7*, 7
checktor_check_result(), *13–17*, *24–52*, *54–72*, *74–93*, *101*
checkup, 9
checkup(), *6*, *11*, *96*
ci_report, 10
ci_report(), *8*
collect_rd_text, 11
collect_rd_text(), *20*
configure_doctor, 12
configure_doctor(), *6*

diagnose_code_issues, 13
diagnose_code_issues(), *5–7*
diagnose_description_issues, 14
diagnose_description_issues(), *5*
diagnose_documentation_issues, 15
diagnose_documentation_issues(), *5*, *7*
diagnose_general_issues, 16
diagnose_general_issues(), *5*, *31*
diagnose_policy_violations, 17
diagnose_policy_violations(), *5*

example_diagnose_scenario, 18
example_diagnose_scenario(), *102*
extract_rd_section, 20
extract_rd_section(), *12*

failed_checks (predicates), 95
find_package_root, 21
find_package_root(), *4*

health_report, 22
health_report(), *6*, *8*, *11*, *99*

is_healthy (predicates), 95
issues, 22
issues(), *5*, *101*

lab_acronyms, 23
lab_authors, 24
lab_authors(), *30*, *37*, *55*
lab_browser_calls, 25
lab_citation_file, 26
lab_code_exercised, 27
lab_commented_examples, 28
lab_core_usage, 29
lab_cph_role, 30
lab_cran_comments_file, 31
lab_cran_comments_file(), *16*
lab_date_format, 32
lab_description_fields, 33
lab_description_file, 34
lab_description_function_quotes, 35
lab_description_length, 36
lab_description_placeholders, 37
lab_description_quoted_quotes, 38
lab_description_quoted_quotes(), *51*, *76*
lab_description_starts_with, 39
lab_detect_cores_robustness, 40
lab_donttest_vs_dontrun, 41
lab_encoding_utf8, 42
lab_example_installs, 43
lab_example_interactive, 44
lab_example_internal_ns, 45

lab_example_internal_ns(), *57*
lab_example_state, 46
lab_example_structure, 47
lab_example_structure(), *44*, *49*
lab_example_tf_usage, 47
lab_example_unparseable, 48
lab_example_writes, 49
lab_file_operations, 50
lab_file_operations(), *50*
lab_format_names, 51
lab_format_names(), *24*, *38*, *58, 59*
lab_globalenv_mod, 52
lab_hardcoded_credentials, 53
lab_home_writing, 54
lab_identifier_format, 55
lab_installed_packages, 56
lab_internal_ns, 57
lab_language_names, 58
lab_language_names(), *24*, *38*, *52*
lab_library_in_pkg, 59
lab_license, 60
lab_license(), *61*
lab_license_file_unneeded, 61
lab_license_year, 62
lab_missing_examples, 63
lab_network_operations, 64
lab_news_file, 65
lab_option_changes, 66
lab_option_changes(), *46*
lab_package_size, 67
lab_print_cat_usage, 68
lab_rd_bibliography, 69
lab_rd_bibliography(), *70*
lab_rd_bibliography_files, 70
lab_rd_bibliography_files(), *69*
lab_readme_links, 71
lab_references, 72
lab_roxygen_usage, 73
lab_seed_setting, 74
lab_seed_setting(), *8*, *19*
lab_software_install, 75
lab_software_install(), *43*
lab_software_names, 76
lab_software_names(), *38*, *51*, *58, 59*
lab_spelling, 77
lab_spelling(), *6*
lab_suggested_in_examples, 78
lab_suggested_in_examples(), *41*
lab_sys_setenv, 80
lab_system_calls, 79
lab_temp_cleanup, 81
lab_tf_usage, 82
lab_tf_usage(), *8*, *19*, *47, 48*
lab_title_case, 83
lab_title_length, 84
lab_title_package_name, 85
lab_title_redundant_phrases, 86
lab_title_starts_with_article, 87
lab_unexported_example_ns, 88
lab_unexported_example_ns(), *45*
lab_url_liveness, 90
lab_url_liveness(), *6*
lab_urls, 89
lab_urls(), *90*
lab_value_tags, 91
lab_version_format, 92
lab_warn_option, 93

n_failed_checks *(predicates)*, 95
n_issues *(predicates)*, 95
not_under_fn_with_call_xpath, 94
not_under_fn_with_call_xpath(), *105*

passed *(predicates)*, 95
passed(), *104*
predicates, 95
prescribe, 97
prescribe(), *6*, *22*, *77, 78*
print.checktor_category_result, 97
print.checktor_check_result, 98
print.checktor_results, 99

read_r_xml, 99
read_r_xml(), *105–107*
register_check, 101
register_check(), *21*, *100*, *105, 106*
registered_checks, 100
registered_checks(), *102*, *106*

show_example_files, 102
summary(), *8*, *96*, *104*
summary.checktor_category_result, 103
summary.checktor_results *(summary.checktor_category_result)*, 103

tidy(), *8*, *96*, *101*

tidy.checktor_category_result
(tidy.checktor_results), 104
tidy.checktor_results, 104
tools::analyze_license(), *60*
tools::check_package_urls(), *90*
tools::parse_Rd(), *11*, *15*, *17*, *20*, *47*, *63*, *91*
tools::toTitleCase(), *83*

undesirable_function_check, 105
undesirable_function_check(), *106*
unregister_check, 105
unregister_check(), *100*, *102*
utils::aspell(), *77*
utils::person(), *31*

xmlparsedata::xml_parse_data(), *99*
xpath_lints, 106
xpath_lints(), *94*, *99*, *100*, *105*, *107*
xpath_per_file, 107
xpath_per_file(), *106*