Package ‘lintr’

October 11, 2022

Title A 'Linter' for R Code

Version 3.0.2

Description Checks adherence to a given style, syntax errors and possible semantic issues. Supports on the fly checking of R code edited with 'RStudio IDE', 'Emacs', 'Vim', 'Sublime Text', 'Atom' and 'Visual Studio Code'.

License MIT + file LICENSE

URL `https://github.com/r-lib/lintr`, `https://lintr.r-lib.org`

BugReports `https://github.com/r-lib/lintr/issues`

Depends R (>= 3.2)

Imports backports,
codetools,
crayon,
cyclocomp,
digest,
glue,
jsonlite,
knitr,
rex,
stats,
utils,
xml2 (>= 1.0.0),
xmlparsedata (>= 1.0.5)

Suggests covr,
httr (>= 1.2.1),
mockery,
patrick,
pkgdown,
rmarkdown,
rstudioapi (>= 0.2),
testthat (>= 3.0.0),
tibble,
withr (>= 2.5.0)

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.1

Collate 'T_and_F_symbol_linter.R'
'utils.R'
'aaa.R'
'actions.R'
'addins.R'
'any_duplicated_linter.R'
'any_is_na_linter.R'
'assignment_linter.R'
'backport_linter.R'
'brace_linter.R'
'cache.R'
'class_equals_linter.R'
'closed_curly_linter.R'
'commas_linter.R'
'comment_linters.R'
'comments.R'
'condition_message_linter.R'
'conjunct_test_linter.R'
'consecutive_stopifnot_linter.R'
'cyclocomp_linter.R'
'declared_functions.R'
'deprecated.R'
'duplicate_argument_linter.R'
'equals_na_linter.R'
'exclude.R'
'expect_comparison_linter.R'
'expect_identical_linter.R'
'expect_length_linter.R'
'expect_lint.R'
'expect_named_linter.R'
'expect_not_linter.R'
'expect_null_linter.R'
'expect_s3_class_linter.R'
'expect_true_false_linter.R'
'expect_type_linter.R'
'extract.R'
'extraction_operator_linter.R'
'fixed_regex_linter.R'
'function_argument_linter.R'
'function_left_parentheses_linter.R'
'get_source_expressions.R'
'ids_with_token.R'
'ifelse_censor_linter.R'
'implicit_integer_linter.R'
'infix_spaces_linter.R'
'inner_combine_linter.R'
'is_lint_level.R'
'line_length_linter.R'
'lint.R'

'linter_tag_docs.R'
'linter_tags.R'
'literal_coercion_linter.R'
'make_linter_from_regex.R'
'methods.R'
'missing_argument_linter.R'
'missing_package_linter.R'
'namespace.R'
'namespace_linter.R'
'nested_ifelse_linter.R'
'no_tab_linter.R'
'numeric_leading_zero_linter.R'
'object_name_linters.R'
'object_usage_linter.R'
'open_curly_linter.R'
'outer_negation_linter.R'
'package_hooks_linter.R'
'paren_body_linter.R'
'paren_brace_linter.R'
'paste_linter.R'
'path_linters.R'
'pipe_call_linter.R'
'pipe_continuation_linter.R'
'redundant_ifelse_linter.R'
'regex_subset_linter.R'
'semicolon_linter.R'
'seq_linter.R'
'settings.R'
'single_quotes_linter.R'
'spaces_inside_linter.R'
'spaces_left_parentheses_linter.R'
'sprintf_linter.R'
'string_boundary_linter.R'
'strings_as_factors_linter.R'
'system_file_linter.R'
'trailing_blank_lines_linter.R'
'trailing_whitespace_linter.R'
'tree-utils.R'
'undesirable_function_linter.R'
'undesirable_operator_linter.R'
'unneeded_concatenation_linter.R'
'unreachable_code_linter.R'
'unused_import_linter.R'
'use_lintr.R'
'vector_logic_linter.R'
'with.R'
'with_id.R'
'xml_nodes_to_lints.R'
'xp_utils.R'
'yoda_test_linter.R'
'zzz.R'

R topics documented:

`absolute_path_linter` *Absolute path linter*

Description

Check that no absolute paths are used (e.g. "/var", "C:\System", "~/docs").

Usage

```
absolute_path_linter(lax = TRUE)
```

Arguments

`lax`	Less stringent linting, leading to fewer false positives. If `TRUE`, only lint path strings, which • contain at least two path elements, with one having at least two characters and • contain only alphanumeric chars (including UTF-8), spaces, and win32-allowed punctuation

Tags

best_practices, configurable, robustness

See Also

linters for a complete list of linters available in lintr.

`all_undesirable_functions` *Default undesirable functions and operators*

Description

Lists of function names and operators for `undesirable_function_linter()` and `undesirable_operator_linter()`. There is a list for the default elements and another that contains all available elements. Use `modify_defaults()` to produce a custom list.

Usage

```
all_undesirable_functions

default_undesirable_functions

all_undesirable_operators

default_undesirable_operators
```

Format

A named list of character strings.

Details

The following functions are sometimes regarded as undesirable:

- `attach()` modifies the global search path. Use roxygen2's @importFrom statement in packages, or `::` in scripts.
- `browser()` pauses execution when run and is likely a leftover from debugging. It should be removed.
- `debug()` traps a function and causes execution to pause when that function is run. It should be removed.
- `debugcall()` works similarly to `debug()`, causing execution to pause. It should be removed.
- `debugonce()` is only useful for interactive debugging. It should be removed.
- `detach()` modifies the global search path. Detaching environments from the search path is rarely necessary in production code.
- `ifelse()` isn't type stable. Use an `if/else` block for scalar logic, or use `dplyr::if_else()/data.table::fifels` for type stable vectorized logic.
- `.libPaths()` permanently modifies the library location. Use `withr::with_libpaths()` for a temporary change instead.
- `library()` modifies the global search path. Use roxygen2's @importFrom statement in packages, or `::` in scripts.
- `loadNamespace()` doesn't provide an easy way to signal failures. Use the return value of `requireNamespace()` instead.
- `mapply()` isn't type stable. Use `Map()` to guarantee a list is returned and simplify accordingly.

- `options()` permanently modifies the session options. Use `withr::with_options()` for a temporary change instead.
- `par()` permanently modifies the graphics device parameters. Use `withr::with_par()` for a temporary change instead.
- `require()` modifies the global search path. Use roxygen2's @importFrom statement in packages, and `library()` or `::` in scripts.
- `sapply()` isn't type stable. Use `vapply()` with an appropriate `FUN.VALUE=` argument to obtain type stable simplification.
- `setwd()` modifies the global working directory. Use `withr::with_dir()` for a temporary change instead.
- `sink()` permanently redirects output. Use `withr::with_sink()` for a temporary redirection instead.
- `source()` loads code into the global environment unless `local = TRUE` is used, which can cause unexpected behaviour.
- `substring()` should be replaced by `substr()` with appropriate `stop=` value.
- `Sys.setenv()` permanently modifies the global environment variables. Use `withr::with_envvar()` for a temporary change instead.
- `Sys.setlocale()` permanently modifies the session locale. Use `withr::with_locale()` for a temporary change instead.
- `trace()` traps a function and causes execution of arbitrary code when that function is run. It should be removed.
- `undebug()` is only useful for interactive debugging with `debug()`. It should be removed.
- `untrace()` is only useful for interactive debugging with `trace()`. It should be removed.

The following operators are sometimes regarded as undesirable:

- `:::` accesses non-exported functions inside packages. Code relying on these is likely to break in future versions of the package because the functions are not part of the public interface and may be changed or removed by the maintainers without notice. Use public functions via `::` instead.
- `<<-` and `->>` assign outside the current environment in a way that can be hard to reason about. Prefer fully-encapsulated functions wherever possible, or, if necessary, assign to a specific environment with `assign()`. Recall that you can create an environment at the desired scope with `new.env()`.

`any_duplicated_linter` *Require usage of anyDuplicated() > 0 over any(duplicated(.))*

Description

`anyDuplicated()` exists as a replacement for `any(duplicated(.))` which is more efficient for simple objects, and in the worst case is the same efficiency. Therefore it should be used in all situations instead of the latter.

Usage

```
any_duplicated_linter()
```

Details

Also match usage like `length(unique(x$col)) == nrow(x)`, which can be replaced by `anyDuplicated(x$col) == 0L`.

Tags

best_practices, efficiency

See Also

linters for a complete list of linters available in lintr.

`any_is_na_linter` *Require usage of anyNA over any(is.na(.))*

Description

`anyNA()` exists as a replacement for `any(is.na(.))` which is more efficient for simple objects, and in the worst case is the same efficiency. Therefore it should be used in all situations instead of the latter.

Usage

```
any_is_na_linter()
```

Tags

best_practices, efficiency

See Also

linters for a complete list of linters available in lintr.

`assignment_linter` *Assignment linter*

Description

Check that `<-` is always used for assignment.

Usage

```
assignment_linter(
  allow_cascading_assign = TRUE,
  allow_right_assign = FALSE,
  allow_trailing = TRUE
)
```

Arguments

`allow_cascading_assign`	Logical, default `TRUE`. If `FALSE`, `<<-` and `->>` are not allowed.
`allow_right_assign`	Logical, default `FALSE`. If `TRUE`, `->` and `->>` are allowed.
`allow_trailing`	Logical, default `TRUE`. If `FALSE` then assignments aren't allowed at end of lines.

Tags

consistency, default, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#assignment-1`

`available_linters` *Get Linter metadata from a package*

Description

`available_linters()` obtains a tagged list of all Linters available in a package.

`available_tags()` searches for available tags.

Usage

```
available_linters(packages = "lintr", tags = NULL, exclude_tags = "deprecated")

available_tags(packages = "lintr")
```

Arguments

`packages`	A character vector of packages to search for linters.
`tags`	Optional character vector of tags to search. Only linters with at least one matching tag will be returned. If `tags` is `NULL`, all linters will be returned.
`exclude_tags`	Tags to exclude from the results. Linters with at least one matching tag will not be returned. If `except_tags` is `NULL`, no linters will be excluded.

Value

`available_linters` returns a data frame with columns 'linter', 'package' and 'tags':

linter A character column naming the function associated with the linter.

package A character column containing the name of the package providing the linter.

tags A list column containing tags associated with the linter.

`available_tags` returns a character vector of linter tags used by the packages.

Package Authors

To implement `available_linters()` for your package, include a file `inst/lintr/linters.csv` in your package. The CSV file must contain the columns 'linter' and 'tags', and be UTF-8 encoded. Additional columns will be silently ignored if present and the columns are identified by name. Each row describes a linter by

1. its function name (e.g. `"assignment_linter"`) in the column 'linter'.
2. space-separated tags associated with the linter (e.g. `"style consistency default"`) in the column 'tags'.

Tags should be snake_case.

See `available_tags("lintr")` to find out what tags are already used by lintr.

See Also

linters for a complete list of linters available in lintr.

Examples

```
lintr_linters <- available_linters()

# If the package doesn't exist or isn't installed, an empty data frame will be returned
available_linters("does-not-exist")

lintr_linters2 <- available_linters(c("lintr", "does-not-exist"))
identical(lintr_linters, lintr_linters2)
available_tags()
```

`backport_linter` *Backport linter*

Description

Check for usage of unavailable functions. Not reliable for testing r-devel dependencies.

Usage

```
backport_linter(r_version = getRversion(), except = character())
```

Arguments

`r_version`	Minimum R version to test for compatibility
`except`	Character vector of functions to be excluded from linting. Use this to list explicitly defined backports, e.g. those imported from the backports package or manually defined in your package.

Tags

configurable, package_development, robustness

See Also

linters for a complete list of linters available in lintr.

best_practices_linters

Best practices linters

Description

Linters checking the use of coding best practices, such as explicit typing of numeric constants.

Linters

The following linters are tagged with 'best_practices':

- absolute_path_linter
- any_duplicated_linter
- any_is_na_linter
- class_equals_linter
- commented_code_linter
- condition_message_linter
- conjunct_test_linter
- cyclocomp_linter
- expect_comparison_linter
- expect_length_linter
- expect_named_linter
- expect_not_linter
- expect_null_linter
- expect_s3_class_linter
- expect_s4_class_linter
- expect_true_false_linter
- expect_type_linter
- extraction_operator_linter
- fixed_regex_linter
- function_argument_linter
- ifelse_censor_linter
- implicit_integer_linter
- literal_coercion_linter
- nonportable_path_linter
- outer_negation_linter
- paste_linter
- redundant_ifelse_linter
- regex_subset_linter
- seq_linter
- system_file_linter

- `T_and_F_symbol_linter`
- `undesirable_function_linter`
- `undesirable_operator_linter`
- `unreachable_code_linter`
- `unused_import_linter`
- `vector_logic_linter`
- `yoda_test_linter`

See Also

linters for a complete list of linters available in lintr.

`brace_linter` *Brace linter*

Description

Perform various style checks related to placement and spacing of curly braces:

Usage

```
brace_linter(allow_single_line = FALSE)
```

Arguments

`allow_single_line`	if `TRUE`, allow an open and closed curly pair on the same line.

Details

- Opening curly braces are never on their own line and are always followed by a newline.
- Opening curly braces have a space before them.
- Closing curly braces are on their own line unless they are followed by an `else`.
- Closing curly braces in `if` conditions are on the same line as the corresponding `else`.
- Either both or neither branch in `if`/`else` use curly braces, i.e., either both branches use `{...}` or neither does.
- Functions spanning multiple lines use curly braces.

Tags

configurable, default, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#indenting`
`https://style.tidyverse.org/syntax.html#if-statements`

`checkstyle_output` *Checkstyle Report for lint results*

Description

Generate a report of the linting results using the Checkstyle XML format.

Usage

```
checkstyle_output(lints, filename = "lintr_results.xml")
```

Arguments

`lints`	the linting results.
`filename`	the name of the output report

`class_equals_linter` *Block comparison of class with ==*

Description

Usage like `class(x) == "character"` is prone to error since class in R is in general a vector. The correct version for S3 classes is `inherits()`: `inherits(x, "character")`. Often, class `k` will have an `is.` equivalent, for example `is.character()` or `is.data.frame()`.

Usage

```
class_equals_linter()
```

Details

Similar reasoning applies for `class(x) %in% "character"`

Tags

best_practices, consistency, robustness

See Also

linters for a complete list of linters available in lintr.

`clear_cache` *Clear the lintr cache*

Description

Clear the lintr cache

Usage

```
clear_cache(file = NULL, path = NULL)
```

Arguments

`file`	filename whose cache to clear. If you pass `NULL`, it will delete all of the caches.
`path`	directory to store caches. Reads option 'lintr.cache_directory' as the default.

Value

0 for success, 1 for failure, invisibly.

`closed_curly_linter` *Closed curly linter*

Description

Check that closed curly braces are on their own line unless they follow an else, comma, or closing bracket.

Usage

```
closed_curly_linter(allow_single_line = FALSE)
```

Arguments

`allow_single_line`	if `TRUE`, allow an open and closed curly pair on the same line.

Tags

configurable, deprecated, readability, style

See Also

linters for a complete list of linters available in lintr.
https://style.tidyverse.org/syntax.html#indenting

commas_linter *Commas linter*

Description

Check that all commas are followed by spaces, but do not have spaces before them.

Usage

```
commas_linter()
```

Tags

default, readability, style

See Also

linters for a complete list of linters available in lintr.
https://style.tidyverse.org/syntax.html#commas

commented_code_linter *Commented code linter*

Description

Check that there is no commented code outside roxygen blocks.

Usage

```
commented_code_linter()
```

Tags

best_practices, default, readability, style

See Also

linters for a complete list of linters available in lintr.

`common_mistakes_linters`
Common mistake linters

Description

Linters highlighting common mistakes, such as duplicate arguments.

Linters

The following linters are tagged with 'common_mistakes':

- `duplicate_argument_linter`
- `equals_na_linter`
- `missing_argument_linter`
- `missing_package_linter`
- `sprintf_linter`
- `unused_import_linter`

See Also

linters for a complete list of linters available in lintr.

`condition_message_linter`
Block usage of paste() and paste0() with messaging functions using ...

Description

`stop(paste0(...))` is strictly redundant – `stop(...)` is equivalent. `stop(...)` is also preferable to `stop(paste(...))`. The same applies to all default condition functions, i.e., `stop()`, `warning()`, `message()`, and `packageStartupMessage()`.

Usage

```
condition_message_linter()
```

Tags

best_practices, consistency

See Also

linters for a complete list of linters available in lintr.

configurable_linters *Configurable linters*

Description

Generic linters which support custom configuration to your needs.

Linters

The following linters are tagged with 'configurable':

- absolute_path_linter
- backport_linter
- brace_linter
- closed_curly_linter
- cyclocomp_linter
- duplicate_argument_linter
- line_length_linter
- missing_argument_linter
- namespace_linter
- nonportable_path_linter
- object_length_linter
- object_name_linter
- open_curly_linter
- semicolon_linter
- semicolon_terminator_linter
- todo_comment_linter
- undesirable_function_linter
- undesirable_operator_linter
- unneeded_concatenation_linter
- unused_import_linter

See Also

linters for a complete list of linters available in lintr.

`conjunct_test_linter` *Force && conditions in expect_true(), expect_false() to be written separately*

Description

For readability of test outputs, testing only one thing per call to `testthat::expect_true()` is preferable, i.e., `expect_true(A); expect_true(B)` is better than `expect_true(A && B)`, and `expect_false(A); expect_false(B)` is better than `expect_false(A || B)`.

Usage

```
conjunct_test_linter(allow_named_stopifnot = TRUE)
```

Arguments

`allow_named_stopifnot`	Logical, `TRUE` by default. If `FALSE`, "named" calls to `stopifnot()`, available since R 4.0.0 to provide helpful messages for test failures, are also linted.

Details

Similar reasoning applies to `&&` usage inside `stopifnot()` and `assertthat::assert_that()` calls.

Tags

best_practices, package_development, readability

See Also

linters for a complete list of linters available in lintr.

`consecutive_stopifnot_linter` *Force consecutive calls to stopifnot into just one when possible*

Description

`stopifnot()` accepts any number of tests, so sequences like `stopifnot(x); stopifnot(y)` are redundant.

Usage

```
consecutive_stopifnot_linter()
```

Tags

consistency, readability, style

See Also

linters for a complete list of linters available in lintr.

`consistency_linters` *Consistency linters*

Description

Linters checking enforcing a consistent alternative if there are multiple syntactically valid ways to write something.

Linters

The following linters are tagged with 'consistency':

- `assignment_linter`
- `class_equals_linter`
- `condition_message_linter`
- `consecutive_stopifnot_linter`
- `function_argument_linter`
- `implicit_integer_linter`
- `inner_combine_linter`
- `literal_coercion_linter`
- `no_tab_linter`
- `numeric_leading_zero_linter`
- `object_name_linter`
- `paste_linter`
- `redundant_ifelse_linter`
- `seq_linter`
- `single_quotes_linter`
- `system_file_linter`
- `T_and_F_symbol_linter`

See Also

linters for a complete list of linters available in lintr.

`correctness_linters` *Correctness linters*

Description

Linters highlighting possible programming mistakes, such as unused variables.

Linters

The following linters are tagged with 'correctness':

- `duplicate_argument_linter`
- `equals_na_linter`
- `missing_argument_linter`
- `namespace_linter`
- `object_usage_linter`
- `package_hooks_linter`
- `sprintf_linter`

See Also

linters for a complete list of linters available in lintr.

`cyclocomp_linter` *Cyclomatic complexity linter*

Description

Check for overly complicated expressions. See `cyclocomp::cyclocomp()`.

Usage

```
cyclocomp_linter(complexity_limit = 15L)
```

Arguments

`complexity_limit`	expressions with a cyclomatic complexity higher than this are linted, defaults to 15. See `cyclocomp::cyclocomp()`.

Tags

best_practices, configurable, default, readability, style

See Also

linters for a complete list of linters available in lintr.

`default_linters` *Default linters*

Description

List of default linters for `lint()`. Use `linters_with_defaults()` to customize it. Most of the default linters are based on the tidyverse style guide.

The set of default linters is as follows (any parameterised linters, eg, `line_length_linter` use their default argument(s), see ?`<linter_name>` for details):

Usage

```
default_linters
```

Format

An object of class `list` of length 24.

Linters

The following linters are tagged with 'default':

- `assignment_linter`
- `brace_linter`
- `commas_linter`
- `commented_code_linter`
- `cyclocomp_linter`
- `equals_na_linter`
- `function_left_parentheses_linter`
- `infix_spaces_linter`
- `line_length_linter`
- `no_tab_linter`
- `object_length_linter`
- `object_name_linter`
- `object_usage_linter`
- `paren_body_linter`
- `pipe_continuation_linter`
- `semicolon_linter`
- `seq_linter`
- `single_quotes_linter`
- `spaces_inside_linter`
- `spaces_left_parentheses_linter`
- `T_and_F_symbol_linter`
- `trailing_blank_lines_linter`
- `trailing_whitespace_linter`
- `vector_logic_linter`

See Also

linters for a complete list of linters available in lintr.

`default_settings` *Default lintr settings*

Description

Default lintr settings

Usage

```
default_settings
```

Format

An object of class `list` of length 12.

See Also

`read_settings()`, default_linters

`deprecated_linters` *Deprecated linters*

Description

Linters that are deprecated and provided for backwards compatiblility only. These linters will be excluded from `linters_with_tags()` by default.

Linters

The following linters are tagged with 'deprecated':

- `closed_curly_linter`
- `open_curly_linter`
- `paren_brace_linter`
- `semicolon_terminator_linter`

See Also

linters for a complete list of linters available in lintr.

duplicate_argument_linter *Duplicate argument linter*

Description

Check for duplicate arguments in function calls.

Usage

```
duplicate_argument_linter(except = character())
```

Arguments

`except`	a character vector of function names as exceptions.

Tags

common_mistakes, configurable, correctness

See Also

linters for a complete list of linters available in lintr.

efficiency_linters *Efficiency linters*

Description

Linters highlighting code efficiency problems, such as unnecessary function calls.

Linters

The following linters are tagged with 'efficiency':

- `any_duplicated_linter`
- `any_is_na_linter`
- `fixed_regex_linter`
- `ifelse_censor_linter`
- `inner_combine_linter`
- `literal_coercion_linter`
- `nested_ifelse_linter`
- `outer_negation_linter`
- `redundant_ifelse_linter`
- `regex_subset_linter`
- `seq_linter`
- `string_boundary_linter`

- undesirable_function_linter
- undesirable_operator_linter
- unneeded_concatenation_linter
- vector_logic_linter

See Also

linters for a complete list of linters available in lintr.

`equals_na_linter` *Equality check with NA linter*

Description

Check for `x == NA` and `x != NA`

Usage

```
equals_na_linter()
```

Tags

common_mistakes, correctness, default, robustness

See Also

linters for a complete list of linters available in lintr.

`exclude` *Exclude lines or files from linting*

Description

Exclude lines or files from linting

Usage

```
exclude(lints, exclusions = settings$exclusions, linter_names = NULL, ...)
```

Arguments

`lints`	that need to be filtered.
`exclusions`	manually specified exclusions
`linter_names`	character vector of names of the active linters, used for parsing inline exclusions.
`...`	additional arguments passed to `parse_exclusions()`

Details

Exclusions can be specified in three different ways.

1. single line in the source file. default: `# nolint`, possibly followed by a listing of linters to exclude. If the listing is missing, all linters are excluded on that line. The default listing format is `# nolint: linter_name, linter2_name.`. There may not be anything between the colon and the line exclusion tag and the listing must be terminated with a full stop (`.`) for the linter list to be respected.
2. line range in the source file. default: `# nolint start`, `# nolint end`. `# nolint start` accepts linter lists in the same form as `# nolint`.
3. exclusions parameter, a named list of files with named lists of linters and lines to exclude them on, a named list of the files and lines to exclude, or just the filenames if you want to exclude the entire file, or the directory names if you want to exclude all files in a directory.

`executing_linters` *Code executing linters*

Description

Linters that evaluate parts of the linted code, such as loading referenced packages. These linters should not be used with untrusted code, and may need dependencies of the linted package or project to be available in order to function correctly.

Linters

The following linters are tagged with 'executing':

- `namespace_linter`
- `object_length_linter`
- `object_name_linter`
- `object_usage_linter`
- `unused_import_linter`

See Also

linters for a complete list of linters available in lintr.

`expect_comparison_linter`
Require usage of expect_gt(x, y) over expect_true(x > y) (and similar)

Description

`testthat::expect_gt()`, `testthat::expect_gte()`, `testthat::expect_lt()`, `testthat::expect_lte()`, and `testthat::expect_equal()` exist specifically for testing comparisons between two objects. `testthat::expect_true()` can also be used for such tests, but it is better to use the tailored function instead.

Usage

```
expect_comparison_linter()
```

Tags

best_practices, package_development

See Also

linters for a complete list of linters available in lintr.

`expect_identical_linter`
Require usage of expect_identical(x, y) where appropriate

Description

At Google, `testthat::expect_identical()` should be the default/go-to function for comparing an output to an expected value. `expect_true(identical(x, y))` is an equivalent but unadvised method of the same test. Further, `testthat::expect_equal()` should only be used when `expect_identical()` is inappropriate, i.e., when `x` and `y` need only be *numerically equivalent* instead of fully identical (in which case, provide the `tolerance=` argument to `expect_equal()` explicitly). This also applies when it's inconvenient to check full equality (e.g., names can be ignored, in which case `ignore_attr = "names"` should be supplied to `expect_equal()` (or, for 2nd edition, `check.attributes = FALSE`).

Usage

```
expect_identical_linter()
```

Exceptions

The linter allows `expect_equal()` in three circumstances:

1. A named argument is set (e.g. `ignore_attr` or `tolerance`)
2. Comparison is made to an explicit decimal, e.g. `expect_equal(x, 1.0)` (implicitly setting `tolerance`)
3. `...` is passed (wrapper functions whcih might set arguments such as `ignore_attr` or `tolerance`)

Tags

package_development

See Also

linters for a complete list of linters available in lintr.

`expect_length_linter` *Require usage of expect_length(x, n) over expect_equal(length(x), n)*

Description

`testthat::expect_length()` exists specifically for testing the `length()` of an object. `testthat::expect_equal()` can also be used for such tests, but it is better to use the tailored function instead.

Usage

```
expect_length_linter()
```

Tags

best_practices, package_development, readability

See Also

linters for a complete list of linters available in lintr.

`expect_lint` *Lint expectation*

Description

This is an expectation function to test that the lints produced by `lint` satisfy a number of checks.

Usage

```
expect_lint(content, checks, ..., file = NULL, language = "en")
```

Arguments

`content` a character vector for the file content to be linted, each vector element representing a line of text.

`checks` checks to be performed:

NULL check that no lints are returned.

single string or regex object check that the single lint returned has a matching message.

named list check that the single lint returned has fields that match. Accepted fields are the same as those taken by `Lint()`.

	list of named lists for each of the multiple lints returned, check that it matches the checks in the corresponding named list (as described in the point above).
	Named vectors are also accepted instead of named lists, but this is a compatibility feature that is not recommended for new code.
`...`	arguments passed to `lint()`, e.g. the linters or cache to use.
`file`	if not `NULL`, read content from the specified file rather than from `content`.
`language`	temporarily override Rs `LANGUAGE` envvar, controlling localisation of base R error messages. This makes testing them reproducible on all systems irrespective of their native R language setting.

Value

`NULL`, invisibly.

Examples

```
# no expected lint
expect_lint("a", NULL, trailing_blank_lines_linter)

# one expected lint
expect_lint("a\n", "superfluous", trailing_blank_lines_linter)
expect_lint("a\n", list(message="superfluous", line_number=2), trailing_blank_lines_linter)

# several expected lints
expect_lint("a\n\n", list("superfluous", "superfluous"), trailing_blank_lines_linter)
expect_lint(
  "a\n\n",
 list(list(message="superfluous", line_number=2), list(message="superfluous", line_number=3)),
  trailing_blank_lines_linter()
)
```

`expect_lint_free` *Test that the package is lint free*

Description

This function is a thin wrapper around lint_package that simply tests there are no lints in the package. It can be used to ensure that your tests fail if the package contains lints.

Usage

```
expect_lint_free(...)
```

Arguments

`...`	arguments passed to `lint_package()`

`expect_named_linter` *Require usage of expect_named(x, n) over expect_equal(names(x), n)*

Description

`testthat::expect_named()` exists specifically for testing the `names()` of an object. `testthat::expect_equal()` can also be used for such tests, but it is better to use the tailored function instead.

Usage

```
expect_named_linter()
```

Tags

best_practices, package_development, readability

See Also

linters for a complete list of linters available in lintr.

`expect_not_linter` *Require usage of expect_false(.) over expect_true(!.)*

Description

`testthat::expect_false()` exists specifically for testing that an output is `FALSE`. `testthat::expect_true()` can also be used for such tests by negating the output, but it is better to use the tailored function instead. The reverse is also true – use `expect_false(A)` instead of `expect_true(!A)`.

Usage

```
expect_not_linter()
```

Tags

best_practices, package_development, readability

See Also

linters for a complete list of linters available in lintr.

`expect_null_linter` *expect_null Linter*

Description

Require usage of `expect_null(x)` over `expect_equal(x, NULL)` and similar usages.

Usage

```
expect_null_linter()
```

Details

`testthat::expect_null()` exists specifically for testing for `NULL` objects. `testthat::expect_equal()`, `testthat::expect_identical()`, and `testthat::expect_true()` can also be used for such tests, but it is better to use the tailored function instead.

Tags

best_practices, package_development

See Also

linters for a complete list of linters available in lintr.

`expect_s3_class_linter` *Require usage of expect_s3_class()*

Description

`testthat::expect_s3_class()` exists specifically for testing the class of S3 objects. `testthat::expect_equal()`, `testthat::expect_identical()`, and `testthat::expect_true()` can also be used for such tests, but it is better to use the tailored function instead.

Usage

```
expect_s3_class_linter()
```

Tags

best_practices, package_development

See Also

linters for a complete list of linters available in lintr.

`expect_s4_class_linter` *Require usage of expect_s4_class(x, k) over expect_true(is(x, k))*

Description

`testthat::expect_s4_class()` exists specifically for testing the class of S4 objects. `testthat::expect_true()` can also be used for such tests, but it is better to use the tailored function instead.

Usage

```
expect_s4_class_linter()
```

Tags

best_practices, package_development

See Also

linters for a complete list of linters available in lintr.

`expect_true_false_linter` *Require usage of expect_true(x) over expect_equal(x, TRUE)*

Description

`testthat::expect_true()` and `testthat::expect_false()` exist specifically for testing the `TRUE`/`FALSE` value of an object. `testthat::expect_equal()` and `testthat::expect_identical()` can also be used for such tests, but it is better to use the tailored function instead.

Usage

```
expect_true_false_linter()
```

Tags

best_practices, package_development, readability

See Also

linters for a complete list of linters available in lintr.

`expect_type_linter` *Require usage of expect_type(x, type) over expect_equal(typeof(x), type)*

Description

`testthat::expect_type()` exists specifically for testing the storage type of objects. `testthat::expect_equal()`, `testthat::expect_identical()`, and `testthat::expect_true()` can also be used for such tests, but it is better to use the tailored function instead.

Usage

```
expect_type_linter()
```

Tags

best_practices, package_development

See Also

linters for a complete list of linters available in lintr.

`extraction_operator_linter` *Extraction operator linter*

Description

Check that the `[[` operator is used when extracting a single element from an object, not `[` (subsetting) nor `$` (interactive use).

Usage

```
extraction_operator_linter()
```

Tags

best_practices, style

See Also

linters for a complete list of linters available in lintr.

`fixed_regex_linter` *Require usage of* `fixed=TRUE` *in regular expressions where appropriate*

Description

Invoking a regular expression engine is overkill for cases when the search pattern only involves static patterns.

Usage

```
fixed_regex_linter()
```

Details

NB: for `stringr` functions, that means wrapping the pattern in `stringr::fixed()`.

NB: This linter is likely not able to distinguish every possible case when a fixed regular expression is preferable, rather it seeks to identify likely cases. It should *never* report false positives, however; please report false positives as an error.

Tags

best_practices, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

`function_argument_linter` *Function argument linter*

Description

Check that arguments with defaults come last in all function declarations, as per the tidyverse design guide.

Usage

```
function_argument_linter()
```

Tags

best_practices, consistency, style

See Also

linters for a complete list of linters available in lintr.
`https://design.tidyverse.org/args-data-details.html`

`function_left_parentheses_linter`
Function left parentheses linter

Description

Check that all left parentheses in a function call do not have spaces before them.

Usage

```
function_left_parentheses_linter()
```

Tags

default, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#parentheses`

`get_source_expressions`
Parsed sourced file from a filename

Description

This object is given as input to each linter

Usage

```
get_source_expressions(filename, lines = NULL)
```

Arguments

`filename`	the file to be parsed.
`lines`	a character vector of lines. If `NULL`, then `filename` will be read.

Details

The file is read in using the `encoding` setting. This setting found by taking the first valid result from the following locations

1. The `encoding` key from the usual lintr configuration settings.
2. The `Encoding` field from a Package `DESCRIPTION` file in a parent directory.
3. The `Encoding` field from an R Project `.Rproj` file in a parent directory.
4. `"UTF-8"` as a fallback.

Value

A list with three components:

expressions a list of n+1 objects. The first n elements correspond to each expression in filename, and consist of a list of 9 elements:

- filename (character)
- line (integer) the line in filename where this expression begins
- column (integer) the column in filename where this expression begins
- lines (named character) vector of all lines spanned by this expression, named with the line number corresponding to filename
- parsed_content (data.frame) as given by utils::getParseData() for this expression
- xml_parsed_content (xml_document) the XML parse tree of this expression as given by xmlparsedata::xml_parse_data()
- content (character) the same as lines as a single string (not split across lines)
- (**Deprecated**) find_line (function) a function for returning lines in this expression
- (**Deprecated**) find_column (function) a similar function for columns

The final element of expressions is a list corresponding to the full file consisting of 6 elements:

- filename (character)
- file_lines (character) the readLines() output for this file
- content (character) for .R files, the same as file_lines; for .Rmd scripts, this is the extracted R source code (as text)
- full_parsed_content (data.frame) as given by utils::getParseData() for the full content
- full_xml_parsed_content (xml_document) the XML parse tree of all expressions as given by xmlparsedata::xml_parse_data()
- terminal_newline (logical) records whether filename has a terminal newline (as determined by readLines() producing a corresponding warning)

error A Lint object describing any parsing error.

lines The readLines() output for this file.

---

ids_with_token *Get parsed IDs by token*

Description

Gets the source IDs (row indices) corresponding to given token.

Usage

```
ids_with_token(source_expression, value, fun = `==`, source_file)

with_id(source_expression, id, source_file)
```

Arguments

`source_expression`	A list of source expressions, the result of a call to `get_source_expressions()`, for the desired filename.
`value`	Character. String corresponding to the token to search for. For example: • "SYMBOL" • "FUNCTION" • "EQ_FORMALS" • "$" • "("
`fun`	For additional flexibility, a function to search for in the `token` column of `parsed_content`. Typically `==` or `%in%`.
`source_file`	(DEPRECATED) Same as `source_expression`. Will be removed.
`id`	Integer. The index corresponding to the desired row of `parsed_content`.

Value

`ids_with_token`: The indices of the `parsed_content` data frame entry of the list of source expressions. Indices correspond to the *rows* where `fun` evaluates to `TRUE` for the `value` in the *token* column.

`with_id`: A data frame corresponding to the row(s) specified in `id`.

Functions

- `with_id()`: Return the row of the `parsed_content` entry of the `[get_source_expressions]()` object. Typically used in conjunction with `ids_with_token` to iterate over rows containing desired tokens.

`ifelse_censor_linter` *Block usage of ifelse where pmin or pmax is more appropriate*

Description

`ifelse(x > M, M, x)` is the same as `pmin(x, M)`, but harder to read and requires several passes over the vector.

Usage

```
ifelse_censor_linter()
```

Details

The same goes for other similar ways to censor a vector, e.g. `ifelse(x <= M, x, M)` is `pmin(x, M)`, `ifelse(x < m, m, x)` is `pmax(x, m)`, and `ifelse(x >= m, x, m)` is `pmax(x, m)`.

Tags

best_practices, efficiency

See Also

linters for a complete list of linters available in lintr.

`implicit_integer_linter` *Implicit integer linter*

Description

Check that integers are explicitly typed using the form `1L` instead of `1`.

Usage

```
implicit_integer_linter()
```

Tags

best_practices, consistency, style

See Also

linters for a complete list of linters available in lintr.

`infix_spaces_linter` *Infix spaces linter*

Description

Check that infix operators are surrounded by spaces. Enforces the corresponding Tidyverse style guide rule; see `https://style.tidyverse.org/syntax.html#infix-operators`.

Usage

```
infix_spaces_linter(exclude_operators = NULL, allow_multiple_spaces = TRUE)
```

Arguments

`exclude_operators`
: Character vector of operators to exlude from consideration for linting. Default is to include the following "low-precedence" operators: +, -, ~, >, >=, <, <=, ==, `!=`, &, &&, `|`, `||`, `<-`, `:=`, `<<-`, `->`, `->>`, `=`, `/`, `*`, and any infix operator (exclude infixes by passing `"%%"`). Note that `<-`, `:=`, and `<<-` are included/excluded as a group (indicated by passing `"<-"`), as are `->` and `->>` (*viz*, `"->"`), and that `=` for assignment and for setting arguments in calls are treated the same.

`allow_multiple_spaces`
: Logical, default `TRUE`. If `FALSE`, usage like `x = 2` will also be linted; excluded by default because such usage can sometimes be used for better code alignment, as is allowed by the style guide.

Tags

default, readability, style

See Also

linters for a complete list of linters available in lintr.
https://style.tidyverse.org/syntax.html#infix-operators

`inner_combine_linter` *Require c() to be applied before relatively expensive vectorized functions*

Description

`as.Date(c(a, b))` is logically equivalent to `c(as.Date(a), as.Date(b))`; ditto for the equivalence of several other vectorized functions like `as.POSIXct()` and math functions like `sin()`. The former is to be preferred so that the most expensive part of the operation (`as.Date()`) is applied only once.

Usage

```
inner_combine_linter()
```

Tags

consistency, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

`is_lint_level` *Is this an expression- or a file-level source object?*

Description

Helper for determining whether the current `source_expression` contains all expressions in the current file, or just a single expression.

Usage

```
is_lint_level(source_expression, level = c("expression", "file"))
```

Arguments

`source_expression`	A parsed expression object, i.e., an element of the object returned by `get_source_expressions()`.
`level`	Which level of expression is being tested? `"expression"` means an individual expression, while `"file"` means all expressions in the current file are available.

`line_length_linter` *Line length linter*

Description

Check that the line length of both comments and code is less than `length`.

Usage

```
line_length_linter(length = 80L)
```

Arguments

`length`	maximum line length allowed.

Tags

configurable, default, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#long-lines`

`lint` *Lint a file, directory, or package*

Description

- `lint()` lints a single file.
- `lint_dir()` lints all files in a directory.
- `lint_package()` lints all likely locations for R files in a package, i.e. `R/`, `tests/`, `inst/`, `vignettes/`, `data-raw/`, and `demo/`.

Usage

```
lint(
  filename,
  linters = NULL,
  ...,
  cache = FALSE,
  parse_settings = TRUE,
  text = NULL
)

lint_dir(
  path = ".",
  ...,
  relative_path = TRUE,
    exclusions = list("renv", "packrat"),
  pattern = rex::rex(".", one_of("Rr"), or("", "html", "md", "nw", "rst", "tex", "txt"),
    end),
    parse_settings = TRUE
)

lint_package(
  path = ".",
  ...,
  relative_path = TRUE,
  exclusions = list("R/RcppExports.R"),
  parse_settings = TRUE
)
```

Arguments

`filename`	either the filename for a file to lint, or a character string of inline R code for linting. The latter (inline data) applies whenever `filename` has a newline character (\n).
`linters`	a named list of linter functions to apply. See linters for a full list of default and available linters.
`...`	Provide additional arguments to be passed to: • `exclude()` (in case of `lint()`; e.g. `lints` or `exclusions`) • `lint()` (in case of `lint_dir()` and `lint_package()`; e.g. `linters` or `cache`)
`cache`	given a logical, toggle caching of lint results. If passed a character string, store the cache in this directory.
`parse_settings`	whether to try and parse the settings.
`text`	Optional argument for supplying a string or lines directly, e.g. if the file is already in memory or linting is being done ad hoc.
`path`	For the base directory of the project (for `lint_dir()`) or package (for `lint_package()`).
`relative_path`	if `TRUE`, file paths are printed using their path relative to the base directory. If `FALSE`, use the full absolute path.
`exclusions`	exclusions for `exclude()`, relative to the package path.
`pattern`	pattern for files, by default it will take files with any of the extensions .R, .Rmd, .Rnw, .Rhtml, .Rrst, .Rtex, .Rtxt allowing for lowercase r (.r, ...)

Details

Read `vignette("lintr")` to learn how to configure which linters are run by default. Note that if files contain unparseable encoding problems, only the encoding problem will be linted to avoid unintelligible error messages from other linters.

Value

A list of lint objects.

Examples

```
## Not run:
  lint("some/file-name.R") # linting a file
  lint("a = 123\n")        # linting inline-code
  lint(text = "a = 123")   # linting inline-code

## End(Not run)

## Not run:
  lint_dir()
  lint_dir(
    linters = list(semicolon_linter())
    exclusions = list("inst/doc/creating_linters.R" = 1, "inst/example/bad.R", "renv")
  )

## End(Not run)
## Not run:
  lint_package()

  lint_package(
    linters = linters_with_defaults(semicolon_linter = semicolon_linter())
    exclusions = list("inst/doc/creating_linters.R" = 1, "inst/example/bad.R")
  )

## End(Not run)
```

---

lint-s3 *Create a* `lint` *object*

Description

Create a `lint` object

Usage

```
Lint(
  filename,
  line_number = 1L,
  column_number = 1L,
  type = c("style", "warning", "error"),
  message = "",
  line = "",
  ranges = NULL,
  linter = ""
)
```

Arguments

`filename`	path to the source file that was linted.
`line_number`	line number where the lint occurred.
`column_number`	column number where the lint occurred.
`type`	type of lint.

`message`	message used to describe the lint error
`line`	code source where the lint occurred
`ranges`	a list of ranges on the line that should be emphasized.
`linter`	deprecated. No longer used.

Value

an object of class 'lint'.

`Linter` *Create a* `linter` *closure*

Description

Create a `linter` closure

Usage

```
Linter(fun, name = linter_auto_name())
```

Arguments

`fun`	A function that takes a source file and returns `lint` objects.
`name`	Default name of the Linter. Lints produced by the linter will be labelled with `name` by default.

Value

The same function with its class set to 'linter'.

`linters` *Available linters*

Description

A variety of linters is available in **lintr**. The most popular ones are readily accessible through `default_linters()`.

Within a `lint()` function call, the linters in use are initialized with the provided arguments and fed with the source file (provided by `get_source_expressions()`).

A data frame of all available linters can be retrieved using `available_linters()`. Documentation for linters is structured into tags to allow for easier discovery; see also `available_tags()`.

Tags

The following tags exist:

- best_practices (37 linters)
- common_mistakes (6 linters)
- configurable (20 linters)
- consistency (17 linters)
- correctness (7 linters)
- default (24 linters)
- deprecated (4 linters)
- efficiency (16 linters)
- executing (5 linters)
- package_development (14 linters)
- readability (37 linters)
- robustness (12 linters)
- style (36 linters)

Linters

The following linters exist:

- `absolute_path_linter` (tags: best_practices, configurable, robustness)
- `any_duplicated_linter` (tags: best_practices, efficiency)
- `any_is_na_linter` (tags: best_practices, efficiency)
- `assignment_linter` (tags: consistency, default, style)
- `backport_linter` (tags: configurable, package_development, robustness)
- `brace_linter` (tags: configurable, default, readability, style)
- `class_equals_linter` (tags: best_practices, consistency, robustness)
- `closed_curly_linter` (tags: configurable, deprecated, readability, style)
- `commas_linter` (tags: default, readability, style)
- `commented_code_linter` (tags: best_practices, default, readability, style)
- `condition_message_linter` (tags: best_practices, consistency)
- `conjunct_test_linter` (tags: best_practices, package_development, readability)
- `consecutive_stopifnot_linter` (tags: consistency, readability, style)
- `cyclocomp_linter` (tags: best_practices, configurable, default, readability, style)
- `duplicate_argument_linter` (tags: common_mistakes, configurable, correctness)
- `equals_na_linter` (tags: common_mistakes, correctness, default, robustness)
- `expect_comparison_linter` (tags: best_practices, package_development)
- `expect_identical_linter` (tags: package_development)
- `expect_length_linter` (tags: best_practices, package_development, readability)
- `expect_named_linter` (tags: best_practices, package_development, readability)
- `expect_not_linter` (tags: best_practices, package_development, readability)
- `expect_null_linter` (tags: best_practices, package_development)

- `expect_s3_class_linter` (tags: best_practices, package_development)
- `expect_s4_class_linter` (tags: best_practices, package_development)
- `expect_true_false_linter` (tags: best_practices, package_development, readability)
- `expect_type_linter` (tags: best_practices, package_development)
- `extraction_operator_linter` (tags: best_practices, style)
- `fixed_regex_linter` (tags: best_practices, efficiency, readability)
- `function_argument_linter` (tags: best_practices, consistency, style)
- `function_left_parentheses_linter` (tags: default, readability, style)
- `ifelse_censor_linter` (tags: best_practices, efficiency)
- `implicit_integer_linter` (tags: best_practices, consistency, style)
- `infix_spaces_linter` (tags: default, readability, style)
- `inner_combine_linter` (tags: consistency, efficiency, readability)
- `line_length_linter` (tags: configurable, default, readability, style)
- `literal_coercion_linter` (tags: best_practices, consistency, efficiency)
- `missing_argument_linter` (tags: common_mistakes, configurable, correctness)
- `missing_package_linter` (tags: common_mistakes, robustness)
- `namespace_linter` (tags: configurable, correctness, executing, robustness)
- `nested_ifelse_linter` (tags: efficiency, readability)
- `no_tab_linter` (tags: consistency, default, style)
- `nonportable_path_linter` (tags: best_practices, configurable, robustness)
- `numeric_leading_zero_linter` (tags: consistency, readability, style)
- `object_length_linter` (tags: configurable, default, executing, readability, style)
- `object_name_linter` (tags: configurable, consistency, default, executing, style)
- `object_usage_linter` (tags: correctness, default, executing, readability, style)
- `open_curly_linter` (tags: configurable, deprecated, readability, style)
- `outer_negation_linter` (tags: best_practices, efficiency, readability)
- `package_hooks_linter` (tags: correctness, package_development, style)
- `paren_body_linter` (tags: default, readability, style)
- `paren_brace_linter` (tags: deprecated, readability, style)
- `paste_linter` (tags: best_practices, consistency)
- `pipe_call_linter` (tags: readability, style)
- `pipe_continuation_linter` (tags: default, readability, style)
- `redundant_ifelse_linter` (tags: best_practices, consistency, efficiency)
- `regex_subset_linter` (tags: best_practices, efficiency)
- `semicolon_linter` (tags: configurable, default, readability, style)
- `semicolon_terminator_linter` (tags: configurable, deprecated, readability, style)
- `seq_linter` (tags: best_practices, consistency, default, efficiency, robustness)
- `single_quotes_linter` (tags: consistency, default, readability, style)
- `spaces_inside_linter` (tags: default, readability, style)
- `spaces_left_parentheses_linter` (tags: default, readability, style)

- `sprintf_linter` (tags: common_mistakes, correctness)
- `string_boundary_linter` (tags: efficiency, readability)
- `strings_as_factors_linter` (tags: robustness)
- `system_file_linter` (tags: best_practices, consistency, readability)
- `T_and_F_symbol_linter` (tags: best_practices, consistency, default, readability, robustness, style)
- `todo_comment_linter` (tags: configurable, style)
- `trailing_blank_lines_linter` (tags: default, style)
- `trailing_whitespace_linter` (tags: default, style)
- `undesirable_function_linter` (tags: best_practices, configurable, efficiency, robustness, style)
- `undesirable_operator_linter` (tags: best_practices, configurable, efficiency, robustness, style)
- `unneeded_concatenation_linter` (tags: configurable, efficiency, readability, style)
- `unreachable_code_linter` (tags: best_practices, readability)
- `unused_import_linter` (tags: best_practices, common_mistakes, configurable, executing)
- `vector_logic_linter` (tags: best_practices, default, efficiency)
- `yoda_test_linter` (tags: best_practices, package_development, readability)

`linters_with_defaults` *Create a linter configuration based on defaults*

Description

Make a new list based on **lintr**'s default linters. The result of this function is meant to be passed to the `linters` argument of `lint()`, or to be put in your configuration file.

Usage

```
linters_with_defaults(..., defaults = default_linters)

with_defaults(..., default = default_linters)
```

Arguments

`...`	Arguments of elements to change. If unnamed, the argument is automatically named. If the named argument already exists in the list of linters, it is replaced by the new element. If it does not exist, it is added. If the value is `NULL`, the linter is removed.
`defaults, default`	Default list of linters to modify. Must be named.

See Also

linters_with_tags for basing off tags attached to linters, possibly across multiple packages. available_linters to get a data frame of available linters. linters for a complete list of linters available in lintr.

Examples

```
# When using interactively you will usually pass the result onto `lint` or `lint_package()`
## Not run:
lint("foo.R", linters = linters_with_defaults(line_length_linter = line_length_linter(120)))

## End(Not run)
# the default linter list with a different line length cutoff
my_linters <- linters_with_defaults(line_length_linter = line_length_linter(120))

# omit the argument name if you are just using different arguments
my_linters <- linters_with_defaults(defaults = my_linters, object_name_linter("camelCase"))

# remove assignment checks (with NULL), add absolute path checks
my_linters <- linters_with_defaults(
  defaults = my_linters,
  assignment_linter = NULL,
  absolute_path_linter()
)
```

`linters_with_tags` *Create a tag-based linter configuration*

Description

Make a new list based on all linters provided by `packages` and tagged with `tags`. The result of this function is meant to be passed to the `linters` argument of `lint()`, or to be put in your configuration file.

Usage

```
linters_with_tags(tags, ..., packages = "lintr", exclude_tags = "deprecated")
```

Arguments

`tags`	Optional character vector of tags to search. Only linters with at least one matching tag will be returned. If `tags` is `NULL`, all linters will be returned.
`...`	Arguments of elements to change. If unnamed, the argument is automatically named. If the named argument already exists in the list of linters, it is replaced by the new element. If it does not exist, it is added. If the value is `NULL`, the linter is removed.
`packages`	A character vector of packages to search for linters.
`exclude_tags`	Tags to exclude from the results. Linters with at least one matching tag will not be returned. If `except_tags` is `NULL`, no linters will be excluded.

Value

A modified list of linters.

See Also

linters_with_defaults for basing off lintr's set of default linters. available_linters to get a data frame of available linters. linters for a complete list of linters available in lintr.

Examples

```
# `linters_with_defaults()` and `linters_with_tags("default")` are the same:
all.equal(linters_with_defaults(), linters_with_tags("default"))

# Get all linters useful for package development
linters_with_tags(tags = "package_development")

# Get all linters provided by lintr
linters_with_tags(tags = NULL)

# Get all linters tagged as "default" from lintr and mypkg
## Not run: linters_with_tags("default", packages = c("lintr", "mypkg"))
```

---

literal_coercion_linter *Require usage of correctly-typed literals over literal coercions*

Description

as.integer(1) (or rlang::int(1)) is the same as 1L but the latter is more concise and gets typed correctly at compilation.

Usage

```
literal_coercion_linter()
```

Details

The same applies to missing sentinels like NA – typically, it is not necessary to specify the storage type of NA, but when it is, prefer using the typed version (e.g. NA_real_) instead of a coercion (like as.numeric(NA)).

Tags

best_practices, consistency, efficiency

See Also

linters for a complete list of linters available in lintr.

---

missing_argument_linter *Missing argument linter*

Description

Check for missing arguments in function calls.

Usage

```
missing_argument_linter(except = c("switch", "alist"), allow_trailing = FALSE)
```

Arguments

`except`	a character vector of function names as exceptions.
`allow_trailing`	always allow trailing empty arguments?

Tags

common_mistakes, configurable, correctness

See Also

linters for a complete list of linters available in lintr.

`missing_package_linter` *Missing package linter*

Description

Check for missing packages in `library()`, `require()`, `loadNamespace()` and `requireNamespace()` calls.

Usage

```
missing_package_linter()
```

Tags

common_mistakes, robustness

See Also

linters for a complete list of linters available in lintr.

`modify_defaults` *Modify lintr defaults*

Description

Modify a list of defaults by name, allowing for replacement, deletion and addition of new elements.

Usage

```
modify_defaults(defaults, ...)
```

Arguments

`defaults`	named list of elements to modify.
`...`	arguments of elements to change. If unnamed, the argument is automatically named. If the named argument already exists in `defaults`, it is replaced by the new element. If it does not exist, it is added. If the value is `NULL`, the element is removed.

Value

A modified list of elements, sorted by name. To achieve this sort in a platform-independent way, two transformations are applied to the names: (1) replace `_` with `0` and (2) convert `tolower()`.

See Also

linters_with_tags, linters_with_defaults for creating linter lists.

Examples

```
# custom list of undesirable functions:
#    remove sapply (using NULL)
#    add cat (with a accompanying message),
#    add print (unnamed, i.e. with no accompanying message)
#    add return (as taken from all_undesirable_functions)
my_undesirable_functions <- modify_defaults(defaults = default_undesirable_functions,
  sapply=NULL, "cat"="No cat allowed", "print", all_undesirable_functions[["return"]])
```

`namespace_linter` *Namespace linter*

Description

Check for missing packages and symbols in namespace calls. Note that using `check_exports=TRUE` or `check_nonexports=TRUE` will load packages used in user code so it could potentially change the global state.

Usage

```
namespace_linter(check_exports = TRUE, check_nonexports = TRUE)
```

Arguments

`check_exports`	Check if `symbol` is exported from `namespace` in `namespace::symbol` calls.
`check_nonexports`	Check if `symbol` exists in `namespace` in `namespace:::symbol` calls.

Tags

configurable, correctness, executing, robustness

See Also

linters for a complete list of linters available in lintr.

`nested_ifelse_linter` *Block usage of nested ifelse() calls*

Description

Calling `ifelse` in nested calls is problematic for two main reasons:

1. It can be hard to read – mapping the code to the expected output for such code can be a messy task/require a lot of mental bandwidth, especially for code that nests more than once
2. It is inefficient – `ifelse` can evaluate *all* of its arguments at both yes and no (see https://stackoverflow.com/q/162751
this issue is exacerbated for nested calls

Usage

```
nested_ifelse_linter()
```

Details

Users can instead rely on a more readable alternative modeled after SQL CASE WHEN statements, such as `data.table::fcase` or `dplyr::case_when`, or use a look-up-and-merge approach (build a mapping table between values and outputs and merge this to the input).

Tags

efficiency, readability

See Also

linters for a complete list of linters available in lintr.

`nonportable_path_linter` *Non-portable path linter*

Description

Check that `file.path()` is used to construct safe and portable paths.

Usage

```
nonportable_path_linter(lax = TRUE)
```

Arguments

`lax`	Less stringent linting, leading to fewer false positives. If `TRUE`, only lint path strings, which • contain at least two path elements, with one having at least two characters and • contain only alphanumeric chars (including UTF-8), spaces, and win32-allowed punctuation

Tags

best_practices, configurable, robustness

See Also

linters for a complete list of linters available in lintr.

`no_tab_linter` *No tab linter*

Description

Check that only spaces are used for indentation, not tabs.

Usage

```
no_tab_linter()
```

Tags

consistency, default, style

See Also

linters for a complete list of linters available in lintr.

`numeric_leading_zero_linter` *Require usage of a leading zero in all fractional numerics*

Description

While .1 and 0.1 mean the same thing, the latter is easier to read due to the small size of the '.' glyph.

Usage

```
numeric_leading_zero_linter()
```

Tags

consistency, readability, style

See Also

linters for a complete list of linters available in lintr.

`object_length_linter` *Object length linter*

Description

Check that object names are not too long. The length of an object name is defined as the length in characters, after removing extraneous parts:

Usage

```
object_length_linter(length = 30L)
```

Arguments

`length`	maximum variable name length allowed.

Details

- generic prefixes for implementations of S3 generics, e.g. `as.data.frame.my_class` has length 8.
- leading `.`, e.g. `.my_hidden_function` has length 18.
- "%%" for infix operators, e.g. `%my_op%` has length 5.
- trailing `<-` for assignment functions, e.g. `my_attr<-` has length 7.

Note that this behavior relies in part on having packages in your Imports available; see the detailed note in `object_name_linter()` for more details.

Tags

configurable, default, executing, readability, style

See Also

linters for a complete list of linters available in lintr.

`object_name_linter` *Object name linter*

Description

Check that object names conform to a naming style. The default naming styles are "snake_case" and "symbols".

Usage

```
object_name_linter(styles = c("snake_case", "symbols"))
```

Arguments

`styles`	A subset of ‘symbols’, ‘CamelCase’, ‘camelCase’, ‘snake_case’, ‘SNAKE_CASE’, ‘dotted.case’, ‘lowercase’, ‘UPPERCASE’. A name should match at least one of these styles.

Details

Note when used in a package, in order to ignore objects imported from other namespaces, this linter will attempt `getNamespaceExports()` whenever an `import(PKG)` or `importFrom(PKG, ...)` statement is found in your NAMESPACE file. If `requireNamespace()` fails (e.g., the package is not yet installed), the linter won't be able to ignore some usages that would otherwise be allowed.

Suppose, for example, you have `import(upstream)` in your NAMESPACE, which makes available its exported S3 generic function `a_really_quite_long_function_name` that you then extend in your package by defining a corresponding method for your class `my_class`. Then, if `upstream` is not installed when this linter runs, a lint will be thrown on this object (even though you don't "own" its full name).

The best way to get lintr to work correctly is to install the package so that it's available in the session where this linter is running.

Tags

configurable, consistency, default, executing, style

See Also

linters for a complete list of linters available in lintr.

`object_usage_linter` *Object usage linter*

Description

Check that closures have the proper usage using `codetools::checkUsage()`. Note that this runs `base::eval()` on the code, so do not use with untrusted code.

Usage

```
object_usage_linter(interpret_glue = TRUE)
```

Arguments

`interpret_glue`	If TRUE, interpret `glue::glue()` calls to avoid false positives caused by local variables which are only used in a glue expression.

Tags

correctness, default, executing, readability, style

See Also

linters for a complete list of linters available in lintr.

`open_curly_linter` *Open curly linter*

Description

Check that opening curly braces are never on their own line and are always followed by a newline.

Usage

```
open_curly_linter(allow_single_line = FALSE)
```

Arguments

`allow_single_line`	if `TRUE`, allow an open and closed curly pair on the same line.

Tags

configurable, deprecated, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#indenting`

`outer_negation_linter` *Require usage of !any(.) over all(!.), !all(.) over any(!.)*

Description

`any(!x)` is logically equivalent to `!any(x)`; ditto for the equivalence of `all(!x)` and `!any(x)`. Negating after aggregation only requires inverting one logical value, and is typically more readable.

Usage

```
outer_negation_linter()
```

Tags

best_practices, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

`package_development_linters` *Package development linters*

Description

Linters useful to package developers, for example for writing consistent tests.

Linters

The following linters are tagged with 'package_development':

- `backport_linter`
- `conjunct_test_linter`
- `expect_comparison_linter`
- `expect_identical_linter`
- `expect_length_linter`
- `expect_named_linter`
- `expect_not_linter`
- `expect_null_linter`
- `expect_s3_class_linter`
- `expect_s4_class_linter`
- `expect_true_false_linter`
- `expect_type_linter`
- `package_hooks_linter`
- `yoda_test_linter`

See Also

linters for a complete list of linters available in lintr.

`package_hooks_linter` *Package hooks linter*

Description

Check various common "gotchas" in `.onLoad()`, `.onAttach()`, `.Last.lib()`, and `.onDetach()` namespace hooks that will cause `R CMD check` issues. See Writing R Extensions for details.

Usage

```
package_hooks_linter()
```

Details

1. `.onLoad()` shouldn't call `cat()`, `message()`, `print()`, `writeLines()`, `packageStartupMessage()`, `require()`, `library()`, or `installed.packages()`.
2. `.onAttach()` shouldn't call `cat()`, `message()`, `print()`, `writeLines()`, `library.dynam()`, `require()`, `library()`, or `installed.packages()`.
3. `.Last.lib()` and `.onDetach()` shouldn't call `library.dynam.unload()`.
4. `.onLoad()` and `.onAttach()` should take two arguments, with names matching `^lib` and `^pkg`; `.Last.lib()` and `.onDetach()` should take one argument with name matching `^lib`.

Tags

correctness, package_development, style

See Also

linters for a complete list of linters available in lintr.

`paren_body_linter` *Parenthesis before body linter*

Description

Check that there is a space between right parenthesis and a body expression.

Usage

```
paren_body_linter()
```

Tags

default, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#parentheses`

paren_brace_linter *Parentheses before brace linter*

Description

Check that there is a space between right parentheses and an opening curly brace.

Usage

```
paren_brace_linter()
```

Tags

deprecated, readability, style

See Also

linters for a complete list of linters available in lintr.

parse_exclusions *read a source file and parse all the excluded lines from it*

Description

read a source file and parse all the excluded lines from it

Usage

```
parse_exclusions(
  file,
  exclude = settings$exclude,
  exclude_start = settings$exclude_start,
  exclude_end = settings$exclude_end,
  exclude_linter = settings$exclude_linter,
  exclude_linter_sep = settings$exclude_linter_sep,
  lines = NULL,
  linter_names = NULL
)
```

Arguments

`file`	R source file
`exclude`	regular expression used to mark lines to exclude
`exclude_start`	regular expression used to mark the start of an excluded range
`exclude_end`	regular expression used to mark the end of an excluded range
`exclude_linter`	regular expression used to capture a list of to-be-excluded linters immediately following a `exclude` or `exclude_start` marker.

`exclude_linter_sep`	regular expression used to split a linter list into indivdual linter names for exclusion.
`lines`	a character vector of the content lines of `file`
`linter_names`	Names of active linters

Value

A possibly named list of excluded lines, possibly for specific linters.

`paste_linter` *Raise lints for several common poor usages of paste()*

Description

The following issues are linted by default by this linter (and each can be turned off optionally):

Usage

```
paste_linter(allow_empty_sep = FALSE, allow_to_string = FALSE)
```

Arguments

`allow_empty_sep`	Logical, default `FALSE`. If `TRUE`, usage of `paste()` with `sep = ""` is not linted.
`allow_to_string`	Logical, default `FALSE`. If `TRUE`, usage of `paste()` and `paste0()` with `collapse = ", "` is not linted.

Details

1. Block usage of `paste()` with `sep = ""`. `paste0()` is a faster, more concise alternative.
2. Block usage of `paste()` or `paste0()` with `collapse = ", "`. `toString()` is a direct wrapper for this, and alternatives like `glue::glue_collapse()` might give better messages for humans.
3. Block usage of `paste0()` that supplies `sep=` – this is not a formal argument to `paste0`, and is likely to be a mistake.

Tags

best_practices, consistency

See Also

linters for a complete list of linters available in lintr.

`pipe_call_linter` *Pipe call linter*

Description

Force explicit calls in magrittr pipes, e.g., `1:3 %>% sum()` instead of `1:3 %>% sum`.

Usage

```
pipe_call_linter()
```

Tags

readability, style

See Also

linters for a complete list of linters available in lintr.

`pipe_continuation_linter` *Pipe continuation linter*

Description

Check that each step in a pipeline is on a new line, or the entire pipe fits on one line.

Usage

```
pipe_continuation_linter()
```

Tags

default, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/pipes.html#long-lines-2`

`readability_linters` *Readability linters*

Description

Linters highlighting readability issues, such as missing whitespace.

Linters

The following linters are tagged with 'readability':

- `brace_linter`
- `closed_curly_linter`
- `commas_linter`
- `commented_code_linter`
- `conjunct_test_linter`
- `consecutive_stopifnot_linter`
- `cyclocomp_linter`
- `expect_length_linter`
- `expect_named_linter`
- `expect_not_linter`
- `expect_true_false_linter`
- `fixed_regex_linter`
- `function_left_parentheses_linter`
- `infix_spaces_linter`
- `inner_combine_linter`
- `line_length_linter`
- `nested_ifelse_linter`
- `numeric_leading_zero_linter`
- `object_length_linter`
- `object_usage_linter`
- `open_curly_linter`
- `outer_negation_linter`
- `paren_body_linter`
- `paren_brace_linter`
- `pipe_call_linter`
- `pipe_continuation_linter`
- `semicolon_linter`
- `semicolon_terminator_linter`
- `single_quotes_linter`
- `spaces_inside_linter`
- `spaces_left_parentheses_linter`

- string_boundary_linter
- system_file_linter
- T_and_F_symbol_linter
- unneeded_concatenation_linter
- unreachable_code_linter
- yoda_test_linter

See Also

linters for a complete list of linters available in lintr.

---

read_settings *Read lintr settings*

Description

Lintr searches for settings for a given source file in the following order.

1. options defined as `linter.setting`.
2. `linter_file` in the same directory
3. `linter_file` in the project directory
4. `linter_file` in the user home directory
5. `default_settings()`

Usage

```
read_settings(filename)
```

Arguments

`filename`	source file to be linted

Details

The default linter_file name is `.lintr` but it can be changed with option `lintr.linter_file`. This file is a dcf file, see `base::read.dcf()` for details.

`redundant_ifelse_linter` *Prevent ifelse() from being used to produce TRUE/FALSE or 1/0*

Description

Expressions like `ifelse(x, TRUE, FALSE)` and `ifelse(x, FALSE, TRUE)` are redundant; just `x` or `!x` suffice in R code where logical vectors are a core data structure. `ifelse(x, 1, 0)` is also `as.numeric(x)`, but even this should only be needed rarely.

Usage

```
redundant_ifelse_linter(allow10 = FALSE)
```

Arguments

`allow10`	Logical, default `FALSE`. If `TRUE`, usage like `ifelse(x, 1, 0)` is allowed, i.e., only usage like `ifelse(x, TRUE, FALSE)` is linted.

Tags

best_practices, consistency, efficiency

See Also

linters for a complete list of linters available in lintr.

`regex_subset_linter` *Require usage of direct methods for subsetting strings via regex.*

Description

Using `value = TRUE` in `grep()` returns the subset of the input that matches the pattern, e.g. `grep("[a-m]", letters, value = TRUE)` will return the first 13 elements (`a` through `m`).

Usage

```
regex_subset_linter()
```

Details

`letters[grep("[a-m]", letters)]` and `letters[grepl("[a-m]", letters)]` both return the same thing, but more circuitously and more verbosely.

The `stringr` package also provides an even more readable alternative, namely `str_subset()`, which should be preferred to versions using `str_detect()` and `str_which()`.

Exceptions

Note that `x[grep(pattern, x)]` and `grep(pattern, x, value = TRUE)` are not *completely* interchangeable when `x` is not character (most commonly, when `x` is a factor), because the output of the latter will be a character vector while the former remains a factor. It still may be preferable to refactor such code, as it may be faster to match the pattern on `levels(x)` and use that to subset instead.

Tags

best_practices, efficiency

See Also

linters for a complete list of linters available in lintr.

`robustness_linters` *Robustness linters*

Description

Linters highlighting code robustness issues, such as possibly wrong edge case behaviour.

Linters

The following linters are tagged with 'robustness':

- `absolute_path_linter`
- `backport_linter`
- `class_equals_linter`
- `equals_na_linter`
- `missing_package_linter`
- `namespace_linter`
- `nonportable_path_linter`
- `seq_linter`
- `strings_as_factors_linter`
- `T_and_F_symbol_linter`
- `undesirable_function_linter`
- `undesirable_operator_linter`

See Also

linters for a complete list of linters available in lintr.

`sarif_output` *SARIF Report for lint results*

Description

Generate a report of the linting results using the SARIF format.

Usage

```
sarif_output(lints, filename = "lintr_results.sarif")
```

Arguments

`lints`	the linting results.
`filename`	the name of the output report

`semicolon_linter` *Semicolon linter*

Description

Check that no semicolons terminate expressions.

Usage

```
semicolon_linter(allow_compound = FALSE, allow_trailing = FALSE)

semicolon_terminator_linter(semicolon = c("compound", "trailing"))
```

Arguments

`allow_compound`	Logical, default `FALSE`. If `TRUE`, "compound" semicolons (e.g. as in `x; y`, i.e., on the same line of code) are allowed.
`allow_trailing`	Logical, default `FALSE`. If `TRUE`, "trailing" semicolons (i.e., those that terminate lines of code) are allowed.
`semicolon`	A character vector defining which semicolons to report: **compound** Semicolons that separate two statements on the same line. **trailing** Semicolons following the last statement on the line.

Tags

configurable, default, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#semicolons`

`seq_linter` *Sequence linter*

Description

This linter checks for `1:length(...)`, `1:nrow(...)`, `1:ncol(...)`, `1:NROW(...)` and `1:NCOL(...)` expressions in base-R, or their usage in conjunction with `seq()` (e.g., `seq(length(...))`, `seq(nrow(...))`, etc.).

Usage

```
seq_linter()
```

Details

Additionally, it checks for `1:n()` (from dplyr) and `1:.N` (from data.table).

These often cause bugs when the right-hand side is zero. It is safer to use `base::seq_len()` or `base::seq_along()` instead.

Tags

best_practices, consistency, default, efficiency, robustness

See Also

linters for a complete list of linters available in lintr.

`single_quotes_linter` *Single quotes linter*

Description

Check that only double quotes are used to delimit string constants.

Usage

```
single_quotes_linter()
```

Tags

consistency, default, readability, style

See Also

linters for a complete list of linters available in lintr.
https://style.tidyverse.org/syntax.html#character-vectors

`spaces_inside_linter` *Spaces inside linter*

Description

Check that parentheses and square brackets do not have spaces directly inside them, i.e., directly following an opening delimiter or directly preceding a closing delimiter.

Usage

```
spaces_inside_linter()
```

Tags

default, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#parentheses`

`spaces_left_parentheses_linter` *Spaces before parentheses linter*

Description

Check that all left parentheses have a space before them unless they are in a function call.

Usage

```
spaces_left_parentheses_linter()
```

Tags

default, readability, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#parentheses`

`sprintf_linter` — `sprintf` *linter*

Description

Check for an inconsistent number of arguments or arguments with incompatible types (for literal arguments) in `sprintf` calls.

Usage

```
sprintf_linter()
```

Tags

common_mistakes, correctness

See Also

linters for a complete list of linters available in lintr.

`strings_as_factors_linter` — *Identify cases where stringsAsFactors should be supplied explicitly*

Description

Designed for code bases written for versions of R before 4.0 seeking to upgrade to R >= 4.0, where one of the biggest pain points will surely be the flipping of the default value of `stringsAsFactors` from `TRUE` to `FALSE`.

Usage

```
strings_as_factors_linter()
```

Details

It's not always possible to tell statically whether the change will break existing code because R is dynamically typed – e.g. in `data.frame(x)` if `x` is a string, this code will be affected, but if `x` is a number, this code will be unaffected. However, in `data.frame(x = 'a')`, the output will unambiguously be affected. We can instead supply `stringsAsFactors = TRUE`, which will make this code backwards-compatible.

See `https://developer.r-project.org/Blog/public/2020/02/16/stringsasfactors/`.

Tags

robustness

See Also

linters for a complete list of linters available in lintr.

`string_boundary_linter` *Require usage of startsWith() and endsWith() over grepl()/substr() versions*

Description

`startsWith()` is used to detect fixed initial substrings; it is more readable and more efficient than equivalents using `grepl()` or `substr()`. c.f. `startsWith(x, "abc")`, `grepl("^abc", x)`, `substr(x, 1L, 3L) == "abc"`.

Usage

```
string_boundary_linter(allow_grepl = FALSE)
```

Arguments

`allow_grepl`	Logical, default `FALSE`. If `TRUE`, usages with `grepl()` are ignored. Some authors may prefer the `NA` input to `FALSE` output conciseness offered by `grepl()`, which doesn't have a direct equivalent with `startsWith()` or `endsWith()`.

Details

Ditto for using `endsWith()` to detect fixed terminal substrings.

Note that there is a difference in behavior between how `grepl()` and `startsWith()` (and `endsWith()`) handle missing values. In particular, for `grepl()`, `NA` inputs are considered `FALSE`, while for `startsWith()`, `NA` inputs have `NA` outputs. That means the strict equivalent of `grepl("^abc", x)` is `!is.na(x) & startsWith(x, "abc")`.

We lint `grepl()` usages by default because the `!is.na()` version is more explicit with respect to `NA` handling – though documented, the way `grepl()` handles missing inputs may be surprising to some readers.

Tags

efficiency, readability

See Also

linters for a complete list of linters available in lintr.

`style_linters` *Style linters*

Description

Linters highlighting code style issues.

Linters

The following linters are tagged with 'style':

- assignment_linter
- brace_linter
- closed_curly_linter
- commas_linter
- commented_code_linter
- consecutive_stopifnot_linter
- cyclocomp_linter
- extraction_operator_linter
- function_argument_linter
- function_left_parentheses_linter
- implicit_integer_linter
- infix_spaces_linter
- line_length_linter
- no_tab_linter
- numeric_leading_zero_linter
- object_length_linter
- object_name_linter
- object_usage_linter
- open_curly_linter
- package_hooks_linter
- paren_body_linter
- paren_brace_linter
- pipe_call_linter
- pipe_continuation_linter
- semicolon_linter
- semicolon_terminator_linter
- single_quotes_linter
- spaces_inside_linter
- spaces_left_parentheses_linter
- T_and_F_symbol_linter
- todo_comment_linter
- trailing_blank_lines_linter
- trailing_whitespace_linter
- undesirable_function_linter
- undesirable_operator_linter
- unneeded_concatenation_linter

See Also

linters for a complete list of linters available in lintr.

`system_file_linter` *Block usage of file.path() with system.file()*

Description

`system.file()` has a `...` argument which, internally, is passed to `file.path()`, so including it in user code is repetitive.

Usage

```
system_file_linter()
```

Tags

best_practices, consistency, readability

See Also

linters for a complete list of linters available in lintr.

`todo_comment_linter` *TODO comment linter*

Description

Check that the source contains no TODO comments (case-insensitive).

Usage

```
todo_comment_linter(todo = c("todo", "fixme"))
```

Arguments

`todo`	Vector of strings that identify TODO comments.

Tags

configurable, style

See Also

linters for a complete list of linters available in lintr.

`trailing_blank_lines_linter` *Trailing blank lines linter*

Description

Check that there are no trailing blank lines in source code.

Usage

```
trailing_blank_lines_linter()
```

Tags

default, style

See Also

linters for a complete list of linters available in lintr.

`trailing_whitespace_linter` *Trailing whitespace linter*

Description

Check that there are no space characters at the end of source lines.

Usage

```
trailing_whitespace_linter(allow_empty_lines = FALSE, allow_in_strings = TRUE)
```

Arguments

`allow_empty_lines`	Suppress lints for lines that contain only whitespace.
`allow_in_strings`	Suppress lints for trailing whitespace in string constants.

Tags

default, style

See Also

linters for a complete list of linters available in lintr.

`T_and_F_symbol_linter` *`T` and `F` symbol linter*

Description

Avoid the symbols `T` and `F` (for `TRUE` and `FALSE`).

Usage

```
T_and_F_symbol_linter()
```

Tags

best_practices, consistency, default, readability, robustness, style

See Also

linters for a complete list of linters available in lintr.
`https://style.tidyverse.org/syntax.html#logical-vectors`

`undesirable_function_linter` *Undesirable function linter*

Description

Report the use of undesirable functions, e.g. `base::return()`, `base::options()`, or `base::sapply()` and suggest an alternative.

Usage

```
undesirable_function_linter(
  fun = default_undesirable_functions,
  symbol_is_undesirable = TRUE
)
```

Arguments

`fun`	Named character vector. `names(fun)` correspond to undesirable functions, while the values give a description of why the function is undesirable. If `NA`, no additional information is given in the lint message. Defaults to default_undesirable_functions. To make small customizations to this list, use `modify_defaults()`.
`symbol_is_undesirable`	Whether to consider the use of an undesirable function name as a symbol undesirable or not.

Tags

best_practices, configurable, efficiency, robustness, style

See Also

linters for a complete list of linters available in lintr.

`undesirable_operator_linter` *Undesirable operator linter*

Description

Report the use of undesirable operators, e.g. `:::` or `<<-` and suggest an alternative.

Usage

```
undesirable_operator_linter(op = default_undesirable_operators)
```

Arguments

`op`	Named character vector. `names(op)` correspond to undesirable operators, while the values give a description of why the operator is undesirable. If `NA`, no additional information is given in the lint message. Defaults to default_undesirable_operators. To make small customizations to this list, use `modify_defaults()`.

Tags

best_practices, configurable, efficiency, robustness, style

See Also

linters for a complete list of linters available in lintr.

`unneeded_concatenation_linter` *Unneeded concatenation linter*

Description

Check that the `c()` function is not used without arguments nor with a single constant.

Usage

```
unneeded_concatenation_linter(allow_single_expression = TRUE)
```

Arguments

`allow_single_expression`	Logical, default `TRUE`. If `FALSE`, one-expression usages of `c()` are always linted, e.g. `c(x)` and `c(matrix(...))`. In some such cases, `c()` is being used for its side-effect of stripping non-name attributes; it is usually preferable to use `as.vector()` to accomplish the same more readably.

Tags

configurable, efficiency, readability, style

See Also

linters for a complete list of linters available in lintr.

`unreachable_code_linter` *Block unreachable code and comments following return statements*

Description

Code after a top-level `return()` or `stop()` can't be reached; typically this is vestigial code left after refactoring or sandboxing code, which is fine for exploration, but shouldn't ultimately be checked in. Comments meant for posterity should be placed *before* the final `return()`.

Usage

```
unreachable_code_linter()
```

Tags

best_practices, readability

See Also

linters for a complete list of linters available in lintr.

`unused_import_linter` *Check that imported packages are actually used*

Description

Check that imported packages are actually used

Usage

```
unused_import_linter(
  allow_ns_usage = FALSE,
  except_packages = c("bit64", "data.table", "tidyverse")
)
```

Arguments

`allow_ns_usage`	Suppress lints for packages only used via namespace. This is `FALSE` by default because `pkg::fun()` doesn't require `library(pkg)`. You can use requireNamespace("pkg") to ensure a package is installed without loading it.
`except_packages`	Character vector of packages that are ignored. These are usually attached for their side effects.

Tags

best_practices, common_mistakes, configurable, executing

See Also

linters for a complete list of linters available in lintr.

`use_lintr` *Use lintr in your project*

Description

Create a minimal lintr config file as a starting point for customization

Usage

```
use_lintr(path = ".", type = c("tidyverse", "full"))
```

Arguments

`path`	Path to project root, where a `.lintr` file should be created. If the `.lintr` file already exists, an error will be thrown.
`type`	What kind of configuration to create? • `tidyverse` creates a minimal lintr config, based on the default linters (`linters_with_defaults` These are suitable for following the tidyverse style guide. • `full` creates a lintr config using all available linters via `linters_with_tags()`.

Value

Path to the generated configuration, invisibly.

See Also

`vignette("lintr")` for detailed introduction to using and configuring lintr.

Examples

```
## Not run:
# use the default set of linters
lintr::use_lintr()
# or try all linters
lintr::use_lintr(type = "full")

# then
lintr::lint_dir()

## End(Not run)
```

`vector_logic_linter` *Enforce usage of scalar logical operators in conditional statements*

Description

Usage of `&` in conditional statements is error-prone and inefficient. `condition` in `if (condition) expr` must always be length-1, in which case `&&` is to be preferred. Ditto for `|` vs. `||`.

Usage

```
vector_logic_linter()
```

Details

This linter covers inputs to `if()` and `while()` conditions and to `testthat::expect_true()` and `testthat::expect_false()`.

Note that because `&` and `|` are generics, it is possible that `&&` / `||` are not perfect substitutes because `&` is doing method dispatch in an incompatible way.

Moreover, be wary of code that may have side effects, most commonly assignments. Consider `if ((a <- foo(x)) | (b <- bar(y))) { ... }` vs. `if ((a <- foo(x)) || (b <- bar(y))) { ... }`. Because `||` exits early, if `a` is `TRUE`, the second condition will never be evaluated and `b` will not be assigned. Such usage is not allowed by the Tidyverse style guide, and the code can easily be refactored by pulling the assignment outside the condition, so using `||` is still preferable.

Tags

best_practices, default, efficiency

See Also

linters for a complete list of linters available in lintr.
https://style.tidyverse.org/syntax.html#if-statements

`xml_nodes_to_lints` *Convert an XML node or nodeset into a Lint*

Description

Convenience function for converting nodes matched by XPath-based linter logic into a `Lint()` object to return.

Usage

```
xml_nodes_to_lints(
  xml,
  source_expression,
  lint_message,
  type = c("style", "warning", "error"),
  column_number_xpath = range_start_xpath,
  range_start_xpath = "number(./@col1)",
  range_end_xpath = "number(./@col2)"
)
```

Arguments

`xml`	An `xml_node` object (to generate one `Lint`) or an `xml_nodeset` object (to generate several `Lints`), e.g. as returned by `xml2::xml_find_all()` or `xml2::xml_find_first()` or a list of `xml_node` objects.
`source_expression`	A source expression object, e.g. as returned typically by `lint()`, or more generally by `get_source_expressions()`.
`lint_message`	The message to be included as the `message` to the `Lint` object. If `lint_message` is a character vector the same length as `xml`, the `i`-th lint will be given the `i`-th message.
`type`	type of lint.
`column_number_xpath`	XPath expression to return the column number location of the lint. Defaults to the start of the range matched by `range_start_xpath`. See details for more information.
`range_start_xpath`	XPath expression to return the range start location of the lint. Defaults to the start of the expression matched by `xml`. See details for more information.
`range_end_xpath`	XPath expression to return the range end location of the lint. Defaults to the end of the expression matched by `xml`. See details for more information.

Details

The location XPaths, `column_number_xpath`, `range_start_xpath` and `range_end_xpath` are evaluated using `xml2::xml_find_num()` and will usually be of the form `"number(./relative/xpath)"`. Note that the location line number cannot be changed and lints spanning multiple lines will ignore `range_end_xpath`. `column_number_xpath` and `range_start_xpath` are assumed to always refer to locations on the starting line of the `xml` node.

Value

For `xml_nodes`, a `lint`. For `xml_nodesets`, `lints` (a list of `lints`).

`yoda_test_linter` *Block obvious "yoda tests"*

Description

Yoda tests use `(expected, actual)` instead of the more common `(actual, expected)`. This is not always possible to detect statically; this linter focuses on the simple case of testing an expression against a literal value, e.g. `(1L, foo(x))` should be `(foo(x), 1L)`.

Usage

```
yoda_test_linter()
```

Tags

best_practices, package_development, readability

See Also

linters for a complete list of linters available in lintr. https://en.wikipedia.org/wiki/Yoda_conditions