Package 'lintr'

November 27, 2025

Title A 'Linter' for R Code

Version 3.3.0-1

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

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

Depends R (>= 4.0)

Imports backports (>= 1.5.0), cli (>= 3.4.0), codetools, digest, glue, knitr, rex, stats, utils, xfun, xml2 (>= 1.0.0), xmlparsedata (>= 1.0.5)

Suggests bookdown, cyclocomp, jsonlite, patrick (>= 0.2.0), rlang, rmarkdown, rstudioapi (>= 0.2), testthat (>= 3.2.1), tibble, tufte, withr (>= 2.5.0)

Enhances data.table

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/Needs/development pkgload, testthat, patrick

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

Collate 'make_linter_from_xpath.R' 'xp_utils.R' 'utils.R' 'AAA.R' 'T_and_F_symbol_linter.R' 'absolute_path_linter.R' 'actions.R' 'addins.R' 'all_equal_linter.R' 'any_duplicated_linter.R' 'any_is_na_linter.R' 'assignment_linter.R' 'backport_linter.R' 'boolean_arithmetic_linter.R' 'brace_linter.R' 'cache.R' 'class_equals_linter.R' 'coalesce_linter.R' 'commas_linter.R' 'commented_code_linter.R' 'comparison_negation_linter.R'

'condition_call_linter.R' 'condition_message_linter.R'
'conjunct_test_linter.R' 'consecutive_assertion_linter.R'
'consecutive_mutate_linter.R' 'cyclocomp_linter.R'
'declared_functions.R' 'deprecated.R' 'download_file_linter.R'
'duplicate_argument_linter.R' 'empty_assignment_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_s4_class_linter.R' 'expect_true_false_linter.R'
'expect_type_linter.R' 'extract.R' 'fixed_regex_linter.R'
'for_loop_index_linter.R' 'function_argument_linter.R'
'function_left_parentheses_linter.R' 'function_return_linter.R'
'get_source_expressions.R' 'ids_with_token.R'
'if_not_else_linter.R' 'if_switch_linter.R'
'ifelse_censor_linter.R' 'implicit_assignment_linter.R'
'implicit_integer_linter.R' 'indentation_linter.R'
'infix_spaces_linter.R' 'inner_combine_linter.R'
'is_lint_level.R' 'is_numeric_linter.R'
'keyword_quote_linter.R' 'length_levels_linter.R'
'length_test_linter.R' 'lengths_linter.R'
'library_call_linter.R' 'line_length_linter.R' 'lint.R'
'linter_tag_docs.R' 'linter_tags.R' 'lintr-deprecated.R'
'lintr-package.R' 'list2df_linter.R' 'list_comparison_linter.R'
'literal_coercion_linter.R' 'make_linter_from_regex.R'
'matrix_apply_linter.R' 'methods.R' 'missing_argument_linter.R'
'missing_package_linter.R' 'namespace.R' 'namespace_linter.R'
'nested_ifelse_linter.R' 'nested_pipe_linter.R'
'nonportable_path_linter.R' 'shared_constants.R'
'nrow_subset_linter.R' 'numeric_leading_zero_linter.R'
'nzchar_linter.R' 'object_length_linter.R'
'object_name_linter.R' 'object_overwrite_linter.R'
'object_usage_linter.R' 'one_call_pipe_linter.R'
'outer_negation_linter.R' 'package_hooks_linter.R'
'paren_body_linter.R' 'paste_linter.R' 'path_utils.R'
'pipe_call_linter.R' 'pipe_consistency_linter.R'
'pipe_continuation_linter.R' 'pipe_return_linter.R'
'print_linter.R' 'quotes_linter.R' 'redundant_equals_linter.R'
'redundant_ifelse_linter.R' 'regex_subset_linter.R'
'rep_len_linter.R' 'repeat_linter.R' 'return_linter.R'
'routine_registration_linter.R' 'sample_int_linter.R'
'scalar_in_linter.R' 'semicolon_linter.R' 'seq_linter.R'
'settings.R' 'settings_utils.R' 'sort_linter.R'
'source_utils.R' 'spaces_inside_linter.R'
'spaces_left_parentheses_linter.R' 'sprintf_linter.R'
'stopifnot_all_linter.R' 'string_boundary_linter.R'
'strings_as_factors_linter.R' 'system_file_linter.R'
'terminal_close_linter.R' 'todo_comment_linter.R'

'trailing_blank_lines_linter.R' 'trailing_whitespace_linter.R'
'tree_utils.R' 'undesirable_function_linter.R'
'undesirable_operator_linter.R'
'unnecessary_concatenation_linter.R'
'unnecessary_lambda_linter.R' 'unnecessary_nesting_linter.R'
'unnecessary_placeholder_linter.R' 'unreachable_code_linter.R'
'unused_import_linter.R' 'use_lintr.R' 'vector_logic_linter.R'
'which_grepl_linter.R' 'whitespace_linter.R' 'with.R'
'with_id.R' 'xml_nodes_to_lints.R' 'xml_utils.R'
'yoda_test_linter.R' 'zzz.R'

Language en-US

NeedsCompilation no

Author Jim Hester [aut],
Florent Angly [aut] (GitHub: fangly),
Russ Hyde [aut],
Michael Chirico [aut, cre] (ORCID:
<https://orcid.org/0000-0003-0787-087X>),
Kun Ren [aut],
Alexander Rosenstock [aut] (GitHub: AshesITR),
Indrajeet Patil [aut] (ORCID: <https://orcid.org/0000-0003-1995-6531>),
Hugo Gruson [aut] (ORCID: <https://orcid.org/0000-0002-4094-1476>)

Maintainer Michael Chirico <michaelchirico4@gmail.com>

Repository CRAN

Date/Publication 2025-11-27 07:30:02 UTC

Contents

`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.
- `nonportable_path_linter()`

Examples

```
# will produce lints
lint(
  text = 'R"(/blah/file.txt)"',
  linters = absolute_path_linter()
)

# okay
lint(
  text = 'R"(./blah)"',
  linters = absolute_path_linter()
)
```

`all_equal_linter` *Warn about invalid usage of* `all.equal()`

Description

`all.equal()` returns `TRUE` in the absence of differences but return a character string (not `FALSE`) in the presence of differences. Usage of `all.equal()` without wrapping it in `isTRUE()` in `if` clauses, or preceded by the negation operator `!`, are thus likely to generate unexpected errors if the compared objects have differences. An alternative is to use `identical()` to compare vector of strings or when exact equality is expected.

Usage

```
all_equal_linter()
```

Tags

common_mistakes, robustness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# lints
lint(
  text = 'if (all.equal(a, b)) message("equal")',
  linters = all_equal_linter()
)

lint(
  text = '!all.equal(a, b)',
  linters = all_equal_linter()
)

lint(
  text = 'isFALSE(all.equal(a, b))',
  linters = all_equal_linter()
)

# okay
lint(
  text = 'if (isTRUE(all.equal(a, b))) message("equal")',
  linters = all_equal_linter()
)

lint(
  text = '!identical(a, b)',
```

```
  linters = all_equal_linter()
)

lint(
  text = "!isTRUE(all.equal(a, b))",
  linters = all_equal_linter()
)
```

`all_linters` *Create a linter configuration based on all available linters*

Description

Create a linter configuration based on all available linters

Usage

```
all_linters(..., packages = "lintr")
```

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.
`packages`	A character vector of packages to search for linters.

See Also

- linters_with_defaults for basing off lintr's set of default linters.
- 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

```
names(all_linters())
```

`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:

- `.libPaths()` As an alternative, use `withr::with_libpaths()` for a temporary change instead of permanently modifying the library location.
- `attach()` As an alternative, use roxygen2's @importFrom statement in packages, or `::` in scripts. `attach()` modifies the global search path.
- `browser()` As an alternative, remove this likely leftover from debugging. It pauses execution when run.
- `debug()` As an alternative, remove this likely leftover from debugging. It traps a function and causes execution to pause when that function is run.
- `debugcall()` As an alternative, remove this likely leftover from debugging. It traps a function and causes execution to pause when that function is run.
- `debugonce()` As an alternative, remove this likely leftover from debugging. It traps a function and causes execution to pause when that function is run.
- `detach()` As an alternative, avoid modifying the global search path. Detaching environments from the search path is rarely necessary in production code.
- `library()` As an alternative, use roxygen2's @importFrom statement in packages and `::` in scripts, instead of modifying the global search path.
- `mapply()` As an alternative, use `Map()` to guarantee a list is returned and simplify accordingly.

- `options()` As an alternative, use `withr::with_options()` for a temporary change instead of permanently modifying the session options.
- `par()` As an alternative, use `withr::with_par()` for a temporary change instead of permanently modifying the graphics device parameters.
- `require()` As an alternative, use roxygen2's @importFrom statement in packages and `library()` or `::` in scripts, instead of modifying the global search path.
- `sapply()` As an alternative, use `vapply()` with an appropriate `FUN.VALUE=` argument to obtain type-stable simplification.
- `setwd()` As an alternative, use `withr::with_dir()` for a temporary change instead of modifying the global working directory.
- `sink()` As an alternative, use `withr::with_sink()` for a temporary redirection instead of permanently redirecting output.
- `source()` As an alternative, manage dependencies through packages. `source()` loads code into the global environment unless `local = TRUE` is used, which can cause hard-to-predict behavior.
- `structure()` As an alternative, Use `class<-`, `names<-`, and `attr<-` to set attributes.
- `Sys.setenv()` As an alternative, use `withr::with_envvar()` for a temporary change instead of permanently modifying global environment variables.
- `Sys.setlocale()` As an alternative, use `withr::with_locale()` for a temporary change instead of permanently modifying the session locale.
- `trace()` As an alternative, remove this likely leftover from debugging. It traps a function and causes execution of arbitrary code when that function is run.
- `undebug()` As an alternative, remove this likely leftover from debugging. It is only useful for interactive debugging with `debug()`.
- `untrace()` As an alternative, remove this likely leftover from debugging. It is only useful for interactive debugging with `trace()`.

The following operators are sometimes regarded as undesirable:

- `->>`. It assigns 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()`.
- `:::`. It 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.
- `<<-`. It assigns 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(x) > 0` *over* `any(duplicated(x))`

Description

`anyDuplicated()` exists as a replacement for `any(duplicated(.))`, which is more efficient for simple objects, and is at worst equally efficient. 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.

Examples

```
# will produce lints
lint(
  text = "any(duplicated(x), na.rm = TRUE)",
  linters = any_duplicated_linter()
)

lint(
  text = "length(unique(x)) == length(x)",
  linters = any_duplicated_linter()
)

lint(
  text = "DT[, uniqueN(col) == .N]",
  linters = any_duplicated_linter()
)

# okay
lint(
  text = "anyDuplicated(x)",
  linters = any_duplicated_linter()
)
```

```
lint(
  text = "anyDuplicated(x) == 0L",
  linters = any_duplicated_linter()
)

lint(
  text = "anyDuplicated(DT, by = 'col') == 0L",
  linters = any_duplicated_linter()
)
```

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

Description

`base::anyNA()` exists as a replacement for `any(is.na(x))` which is more efficient for simple objects, and is at worst equally efficient. 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.

Examples

```
# will produce lints
lint(
  text = "any(is.na(x), na.rm = TRUE)",
  linters = any_is_na_linter()
)

lint(
  text = "any(is.na(foo(x)))",
  linters = any_is_na_linter()
)

# okay
lint(
  text = "anyNA(x)",
  linters = any_is_na_linter()
```

```
)

lint(
  text = "anyNA(foo(x))",
  linters = any_is_na_linter()
)

lint(
  text = "any(!is.na(x), na.rm = TRUE)",
  linters = any_is_na_linter()
)
```

`assignment_linter` *Assignment linter*

Description

Check that the specified operator is used for assignment.

Usage

```
assignment_linter(
  operator = c("<-", "<<-"),
  allow_cascading_assign = NULL,
  allow_right_assign = NULL,
  allow_trailing = TRUE,
  allow_pipe_assign = NULL
)
```

Arguments

`operator`	Character vector of valid assignment operators. Defaults to allowing `<-` and `<<-`; other valid options are `=`, `->`, `->>`, `%<>%`; use `"any"` to denote "allow all operators", in which case this linter only considers `allow_trailing` for generating lints.
`allow_cascading_assign`, `allow_right_assign`, `allow_pipe_assign`	(Defunct)
`allow_trailing`	Logical, default `TRUE`. If `FALSE` then assignments aren't allowed at end of lines.

Tags

configurable, consistency, default, style

See Also

- linters for a complete list of linters available in lintr.
- https://style.tidyverse.org/syntax.html#assignment-1
- https://style.tidyverse.org/pipes.html#assignment-2

Examples

```
# will produce lints
lint(
  text = "x = mean(x)",
  linters = assignment_linter()
)

code_lines <- "1 -> x\n2 ->> y"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = assignment_linter()
)

lint(
  text = "x %<>% as.character()",
  linters = assignment_linter()
)

lint(
  text = "x <- 1",
  linters = assignment_linter(operator = "=")
)

# okay
lint(
  text = "x <- mean(x)",
  linters = assignment_linter()
)

code_lines <- "x <- 1\ny <<- 2"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = assignment_linter()
)

# customizing using arguments
code_lines <- "1 -> x\n2 ->> y"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = assignment_linter(operator = "->")
)

lint(
  text = "x <<- 1",
  linters = assignment_linter(operator = "<-")
)

writeLines("foo(bar = \n 1)")
lint(
```

```
  text = "foo(bar = \n 1)",
  linters = assignment_linter(allow_trailing = FALSE)
)

lint(
  text = "x %<>% as.character()",
  linters = assignment_linter(operator = "%<>%")
)

lint(
  text = "x = 1",
  linters = assignment_linter(operator = "=")
)
```

`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. See `available_tags("lintr")` to find out what tags are already used by lintr.
`exclude_tags`	Tags to exclude from the results. Linters with at least one matching tag will not be returned. If `exclude_tags` is `NULL`, no linters will be excluded. Note that `tags` takes priority, meaning that any tag found in both `tags` and `exclude_tags` will be included, not excluded. Note that linters with tag `"defunct"` (which do not work and can no longer be run) cannot be queried directly. See lintr-deprecated instead.

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.
- `available_tags()` to retrieve the set of valid tags.

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. Defaults to the R version currently in use. The version can be specified as a version number, or as a version alias (such as "devel", "oldrel", "oldrel-1").
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.

Examples

```
# will produce lints
lint(
  text = "trimws(x)",
  linters = backport_linter("3.0.0")
)

lint(
  text = "str2lang(x)",
  linters = backport_linter("3.2.0")
)

lint(
  text = "deparse1(expr)",
  linters = backport_linter("3.6.0")
)

# okay
lint(
  text = "trimws(x)",
  linters = backport_linter("3.6.0")
)

lint(
  text = "str2lang(x)",
  linters = backport_linter("3.2.0", except = "str2lang")
)

# Version aliases instead of numbers can also be passed to `r_version`
lint(
  text = "deparse1(expr)",
  linters = backport_linter("release")
)
```

`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
- boolean_arithmetic_linter
- class_equals_linter
- coalesce_linter
- commented_code_linter
- condition_call_linter
- condition_message_linter
- conjunct_test_linter
- cyclocomp_linter
- download_file_linter
- empty_assignment_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
- fixed_regex_linter
- for_loop_index_linter
- function_argument_linter
- function_return_linter
- if_switch_linter

- ifelse_censor_linter
- implicit_assignment_linter
- implicit_integer_linter
- is_numeric_linter
- length_levels_linter
- lengths_linter
- library_call_linter
- list_comparison_linter
- literal_coercion_linter
- nonportable_path_linter
- nrow_subset_linter
- nzchar_linter
- object_overwrite_linter
- outer_negation_linter
- paste_linter
- pipe_return_linter
- print_linter
- redundant_equals_linter
- redundant_ifelse_linter
- regex_subset_linter
- rep_len_linter
- routine_registration_linter
- scalar_in_linter
- seq_linter
- sort_linter
- stopifnot_all_linter
- system_file_linter
- T_and_F_symbol_linter
- terminal_close_linter
- undesirable_function_linter
- undesirable_operator_linter
- unnecessary_lambda_linter
- unnecessary_nesting_linter
- unnecessary_placeholder_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.

`boolean_arithmetic_linter` *Require usage of boolean operators over equivalent arithmetic*

Description

`length(which(x == y)) == 0` is the same as `!any(x == y)`, but the latter is more readable and more efficient.

Usage

```
boolean_arithmetic_linter()
```

Tags

best_practices, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "length(which(x == y)) == 0L",
  linters = boolean_arithmetic_linter()
)

lint(
  text = "sum(grepl(pattern, x)) == 0",
  linters = boolean_arithmetic_linter()
)

# okay
lint(
  text = "!any(x == y)",
  linters = boolean_arithmetic_linter()
)

lint(
  text = "!any(grepl(pattern, x))",
  linters = boolean_arithmetic_linter()
)
```

`brace_linter` *Brace linter*

Description

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

Usage

```
brace_linter(
  allow_single_line = FALSE,
  function_bodies = c("multi_line", "always", "not_inline", "never")
)
```

Arguments

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

`function_bodies`
: When to require function bodies to be wrapped in curly braces. One of

  - `"always"` to require braces around all function bodies, including inline functions,
  - `"not_inline"` to require braces when a function body does not start on the same line as its signature,
  - `"multi_line"` (the default) to require braces when a function definition spans multiple lines,
  - `"never"` to never require braces in function bodies.

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.
- Function bodies are wrapped in 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

Examples

```
# will produce lints
lint(
  text = "f <- function() { 1 }",
  linters = brace_linter()
)

writeLines("if (TRUE) {\n return(1) }")
lint(
  text = "if (TRUE) {\n return(1) }",
  linters = brace_linter()
)

# okay
writeLines("f <- function() {\n  1\n}")
lint(
  text = "f <- function() {\n  1\n}",
  linters = brace_linter()
)

writeLines("if (TRUE) { \n return(1) \n}")
lint(
  text = "if (TRUE) { \n return(1) \n}",
  linters = brace_linter()
)

# customizing using arguments
writeLines("if (TRUE) { return(1) }")
lint(
  text = "if (TRUE) { return(1) }",
  linters = brace_linter(allow_single_line = TRUE)
)
```

---

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.

Examples

```
# will produce lints
lint(
  text = 'is_lm <- class(x) == "lm"',
  linters = class_equals_linter()
)

lint(
  text = 'if ("lm" %in% class(x)) is_lm <- TRUE',
  linters = class_equals_linter()
)

# okay
lint(
  text = 'is_lm <- inherits(x, "lm")',
  linters = class_equals_linter()
)
```

```
lint(
  text = 'if (inherits(x, "lm")) is_lm <- TRUE',
  linters = class_equals_linter()
)
```

`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.

`coalesce_linter` *Encourage usage of the null coalescing operator* `%||%`

Description

The `x %||% y` is equivalent to `if (is.null(x)) y else x`, but more expressive. It is exported by R since 4.4.0, and equivalents have been available in other tidyverse packages for much longer, e.g. 2008 for ggplot2.

Usage

```
coalesce_linter()
```

Tags

best_practices, consistency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "if (is.null(x)) y else x",
  linters = coalesce_linter()
)

lint(
  text = "if (!is.null(x)) x else y",
  linters = coalesce_linter()
)

lint(
  text = "if (is.null(x[1])) x[2] else x[1]",
  linters = coalesce_linter()
)

# okay
lint(
  text = "x %||% y",
  linters = coalesce_linter()
)

lint(
  text = "x %||% y",
  linters = coalesce_linter()
)

lint(
  text = "x[1] %||% x[2]",
  linters = coalesce_linter()
)
```

`commas_linter` *Commas linter*

Description

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

Usage

```
commas_linter(allow_trailing = FALSE)
```

Arguments

`allow_trailing`	If `TRUE`, the linter allows a comma to be followed directly by a closing bracket without a space.

Tags

configurable, default, readability, style

See Also

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

Examples

```
# will produce lints
lint(
  text = "switch(op , x = foo, y = bar)",
  linters = commas_linter()
)

lint(
  text = "mean(x,trim = 0.2,na.rm = TRUE)",
  linters = commas_linter()
)

lint(
  text = "x[ ,, drop=TRUE]",
  linters = commas_linter()
)

lint(
  text = "x[1,]",
  linters = commas_linter()
)

# okay
lint(
  text = "switch(op, x = foo, y = bar)",
  linters = commas_linter()
)

lint(
  text = "switch(op, x = , y = bar)",
  linters = commas_linter()
)

lint(
  text = "mean(x, trim = 0.2, na.rm = TRUE)",
  linters = commas_linter()
)

lint(
  text = "a[1, , 2, , 3]",
  linters = commas_linter()
)
```

```
lint(
  text = "x[1,]",
  linters = commas_linter(allow_trailing = TRUE)
)
```

`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.

Examples

```
# will produce lints
lint(
  text = "# x <- 1",
  linters = commented_code_linter()
)

lint(
  text = "x <- f() # g()",
  linters = commented_code_linter()
)

lint(
  text = "x + y # + z[1, 2]",
  linters = commented_code_linter()
)

# okay
lint(
  text = "x <- 1; x <- f(); x + y",
  linters = commented_code_linter()
)

lint(
```

```
  text = "#' x <- 1",
  linters = commented_code_linter()
)
```

`common_mistakes_linters` *Common mistake linters*

Description

Linters highlighting common mistakes, such as duplicate arguments.

Linters

The following linters are tagged with 'common_mistakes':

- `all_equal_linter`
- `download_file_linter`
- `duplicate_argument_linter`
- `equals_na_linter`
- `length_test_linter`
- `list_comparison_linter`
- `missing_argument_linter`
- `missing_package_linter`
- `pipe_return_linter`
- `redundant_equals_linter`
- `sprintf_linter`
- `unused_import_linter`
- `vector_logic_linter`

See Also

linters for a complete list of linters available in lintr.

`comparison_negation_linter`

Block usages like !(x == y) where a direct relational operator is appropriate

Description

`!(x == y)` is more readably expressed as `x != y`. The same is true of other negations of simple comparisons like `!(x > y)` and `!(x <= y)`.

Usage

```
comparison_negation_linter()
```

Tags

consistency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "!x == 2",
  linters = comparison_negation_linter()
)

lint(
  text = "!(x > 2)",
  linters = comparison_negation_linter()
)

# okay
lint(
  text = "!(x == 2 & y > 2)",
  linters = comparison_negation_linter()
)

lint(
  text = "!(x & y)",
  linters = comparison_negation_linter()
)

lint(
  text = "x != 2",
  linters = comparison_negation_linter()
```

```
)
```

`condition_call_linter` *Recommend usage of* `call. = FALSE` *in conditions*

Description

This linter, with the default `display_call = FALSE`, enforces the recommendation of the tidyverse design guide regarding displaying error calls.

Usage

```
condition_call_linter(display_call = FALSE)
```

Arguments

`display_call`	Logical specifying expected behavior regarding `call.` argument in conditions. • `NA` forces providing `call. =` but ignores its value (this can be used in cases where you expect a mix of `call. = FALSE` and `call. = TRUE`) • `TRUE` lints `call. = FALSE` • `FALSE` forces `call. = FALSE` (lints `call. = TRUE` or missing `call. =` value)

Tags

best_practices, configurable, style, tidy_design

See Also

- linters for a complete list of linters available in lintr.
- https://design.tidyverse.org/err-call.html>

Examples

```
# will produce lints
lint(
  text = "stop('test')",
  linters = condition_call_linter()
)

lint(
  text = "stop('test', call. = TRUE)",
  linters = condition_call_linter()
)

lint(
  text = "stop('test', call. = FALSE)",
  linters = condition_call_linter(display_call = TRUE)
```

```
)

lint(
  text = "stop('this is a', 'test', call. = FALSE)",
  linters = condition_call_linter(display_call = TRUE)
)

# okay
lint(
  text = "stop('test', call. = FALSE)",
  linters = condition_call_linter()
)

lint(
  text = "stop('this is a', 'test', call. = FALSE)",
  linters = condition_call_linter()
)

lint(
  text = "stop('test', call. = TRUE)",
  linters = condition_call_linter(display_call = TRUE)
)
```

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

Description

This linter discourages combining condition functions like `stop()` with string concatenation functions `base::paste()` and `base::paste0()`. This is because

- `stop(paste0(...))` is redundant as it is exactly equivalent to `stop(...)`
- `stop(paste(...))` is similarly equivalent to `stop(...)` with separators (see examples)

The same applies to the other default condition functions as well, i.e., `warning()`, `message()`, and `packageStartupMessage()`.

Usage

```
condition_message_linter()
```

Tags

best_practices, consistency

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = 'stop(paste("a string", "another"))',
  linters = condition_message_linter()
)

lint(
  text = 'warning(paste0("a string", " another"))',
  linters = condition_message_linter()
)

# okay
lint(
  text = 'stop("a string", " another")',
  linters = condition_message_linter()
)

lint(
  text = 'warning("a string", " another")',
  linters = condition_message_linter()
)

lint(
  text = 'warning(paste("a string", "another", sep = "-"))',
  linters = condition_message_linter()
)
```

---

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
- assignment_linter
- backport_linter
- brace_linter

- commas_linter
- condition_call_linter
- conjunct_test_linter
- consecutive_mutate_linter
- cyclocomp_linter
- duplicate_argument_linter
- fixed_regex_linter
- if_not_else_linter
- if_switch_linter
- implicit_assignment_linter
- implicit_integer_linter
- indentation_linter
- infix_spaces_linter
- library_call_linter
- line_length_linter
- missing_argument_linter
- namespace_linter
- nested_pipe_linter
- nonportable_path_linter
- object_length_linter
- object_name_linter
- object_overwrite_linter
- object_usage_linter
- paste_linter
- pipe_consistency_linter
- quotes_linter
- redundant_ifelse_linter
- return_linter
- scalar_in_linter
- semicolon_linter
- string_boundary_linter
- todo_comment_linter
- trailing_whitespace_linter
- undesirable_function_linter
- undesirable_operator_linter
- unnecessary_concatenation_linter
- unnecessary_lambda_linter
- unnecessary_nesting_linter
- unreachable_code_linter
- unused_import_linter

See Also

linters for a complete list of linters available in lintr.

`conjunct_test_linter` *Force* `&&` *conditions to be written separately where appropriate*

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,
  allow_filter = c("never", "not_dplyr", "always")
)
```

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.
`allow_filter`	Character naming the method for linting calls to `filter()`. The default, `"never"`, means `filter()` and `dplyr::filter()` calls are linted; `"not_dplyr"` means only `dplyr::filter()` calls are linted; and `"always"` means no calls to `filter()` are linted. Calls like `stats::filter()` are never linted.

Details

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

Relatedly, `dplyr::filter(DF, A & B)` is the same as `dplyr::filter(DF, A, B)`, but the latter will be more readable / easier to format for long conditions. Note that this linter assumes usages of `filter()` are `dplyr::filter()`; if you're using another function named `filter()`, e.g. `stats::filter()`, please namespace-qualify it to avoid false positives. You can omit linting `filter()` expressions altogether via `allow_filter = TRUE`.

Tags

best_practices, configurable, package_development, pkg_testthat, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "expect_true(x && y)",
  linters = conjunct_test_linter()
)

lint(
  text = "expect_false(x || (y && z))",
  linters = conjunct_test_linter()
)

lint(
 text = "stopifnot('x must be a logical scalar' = length(x) == 1 && is.logical(x) && !is.na(x))",
  linters = conjunct_test_linter(allow_named_stopifnot = FALSE)
)

lint(
  text = "dplyr::filter(mtcars, mpg > 20 & vs == 0)",
  linters = conjunct_test_linter()
)

lint(
  text = "filter(mtcars, mpg > 20 & vs == 0)",
  linters = conjunct_test_linter()
)

# okay
lint(
  text = "expect_true(x || (y && z))",
  linters = conjunct_test_linter()
)

lint(
 text = 'stopifnot("x must be a logical scalar" = length(x) == 1 && is.logical(x) && !is.na(x))',
  linters = conjunct_test_linter(allow_named_stopifnot = TRUE)
)

lint(
  text = "dplyr::filter(mtcars, mpg > 20 & vs == 0)",
  linters = conjunct_test_linter(allow_filter = "always")
)

lint(
  text = "filter(mtcars, mpg > 20 & vs == 0)",
  linters = conjunct_test_linter(allow_filter = "not_dplyr")
)

lint(
  text = "stats::filter(mtcars$cyl, mtcars$mpg > 20 & mtcars$vs == 0)",
  linters = conjunct_test_linter()
)
```

`consecutive_assertion_linter` *Force consecutive calls to assertions into just one when possible*

Description

`base::stopifnot()` accepts any number of tests, so sequences like `stopifnot(x); stopifnot(y)` are redundant. Ditto for tests using `assertthat::assert_that()` without specifying `msg=`.

Usage

```
consecutive_assertion_linter()
```

Tags

consistency, readability, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "stopifnot(x); stopifnot(y)",
  linters = consecutive_assertion_linter()
)

lint(
  text = "assert_that(x); assert_that(y)",
  linters = consecutive_assertion_linter()
)

# okay
lint(
  text = "stopifnot(x, y)",
  linters = consecutive_assertion_linter()
)

lint(
  text = 'assert_that(x, msg = "Bad x!"); assert_that(y)',
  linters = consecutive_assertion_linter()
)
```

`consecutive_mutate_linter` *Require consecutive calls to mutate() to be combined when possible*

Description

`dplyr::mutate()` accepts any number of columns, so sequences like `DF %>% dplyr::mutate(..1) %>% dplyr::mutate(..2)` are redundant – they can always be expressed with a single call to `dplyr::mutate()`.

Usage

```
consecutive_mutate_linter(invalid_backends = "dbplyr")
```

Arguments

`invalid_backends`
: Character vector of packages providing dplyr backends which may not be compatible with combining `mutate()` calls in all cases. Defaults to `"dbplyr"` since not all SQL backends can handle re-using a variable defined in the same `mutate()` expression.

Details

An exception is for some SQL back-ends, where the translation logic may not be as sophisticated as that in the default `dplyr`, for example in `DF %>% mutate(a = a + 1) %>% mutate(b = a - 2)`.

Tags

configurable, consistency, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x %>% mutate(a = 1) %>% mutate(b = 2)",
  linters = consecutive_mutate_linter()
)

# okay
lint(
  text = "x %>% mutate(a = 1, b = 2)",
  linters = consecutive_mutate_linter()
)
```

```
code <- "library(dbplyr)\nx %>% mutate(a = 1) %>% mutate(a = a + 1)"
writeLines(code)
lint(
  text = code,
  linters = consecutive_mutate_linter()
)
```

`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`
- `coalesce_linter`
- `comparison_negation_linter`
- `condition_message_linter`
- `consecutive_assertion_linter`
- `consecutive_mutate_linter`
- `function_argument_linter`
- `if_not_else_linter`
- `if_switch_linter`
- `implicit_integer_linter`
- `inner_combine_linter`
- `is_numeric_linter`
- `keyword_quote_linter`
- `length_levels_linter`
- `literal_coercion_linter`
- `nested_pipe_linter`
- `nrow_subset_linter`
- `numeric_leading_zero_linter`
- `nzchar_linter`
- `object_name_linter`

- paste_linter
- print_linter
- quotes_linter
- redundant_ifelse_linter
- rep_len_linter
- scalar_in_linter
- seq_linter
- system_file_linter
- T_and_F_symbol_linter
- unnecessary_nesting_linter
- which_grepl_linter
- whitespace_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()` function from `{cyclocomp}`.

Usage

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

Arguments

`complexity_limit`
: Maximum cyclomatic complexity, default 15. Expressions more complex than this are linted.

Tags

best_practices, configurable, readability, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "if (TRUE) 1 else 2",
  linters = cyclocomp_linter(complexity_limit = 1L)
)

# okay
lint(
  text = "if (TRUE) 1 else 2",
  linters = cyclocomp_linter(complexity_limit = 2L)
)
```

`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 parameterized linters, e.g., `line_length_linter` use their default argument(s), see `?<linter_name>` for details):

Usage

```
default_linters
```

Format

An object of class `list` of length 26.

Linters

The following linters are tagged with 'default':

- `assignment_linter`
- `brace_linter`
- `commas_linter`
- `commented_code_linter`
- `equals_na_linter`
- `function_left_parentheses_linter`
- `indentation_linter`
- `infix_spaces_linter`
- `line_length_linter`
- `object_length_linter`
- `object_name_linter`
- `object_usage_linter`
- `paren_body_linter`
- `pipe_consistency_linter`
- `pipe_continuation_linter`
- `quotes_linter`
- `return_linter`
- `semicolon_linter`
- `seq_linter`

- spaces_inside_linter
- spaces_left_parentheses_linter
- T_and_F_symbol_linter
- trailing_blank_lines_linter
- trailing_whitespace_linter
- vector_logic_linter
- whitespace_linter

See Also

linters for a complete list of linters available in lintr.

`default_settings` *Default lintr settings*

Description

The default settings consist of

- `linters`: a list of default linters (see `default_linters()`)
- `encoding`: the character encoding assumed for the file
- `exclude`: pattern used to exclude a line of code
- `exclude_start`, `exclude_end`: patterns used to mark start and end of the code block to exclude
- `exclude_linter`, `exclude_linter_sep`: patterns used to exclude linters
- `exclusions`: a list of exclusions, see `exclude()` for a complete description of valid values.
- `cache_directory`: location of cache directory
- `error_on_lint`: decides if error should be produced when any lints are found

There are no settings without defaults, i.e., this list describes every valid setting.

Usage

```
default_settings
```

Format

An object of class `list` of length 11.

See Also

`read_settings()`, default_linters

Examples

```
# available settings
names(default_settings)

# linters included by default
names(default_settings$linters)

# default values for a few of the other settings
default_settings[c(
  "encoding",
  "exclude",
  "exclude_start",
  "exclude_end",
  "exclude_linter",
  "exclude_linter_sep",
  "exclusions",
  "error_on_lint"
)]
```

---

deprecated_linters *Deprecated linters*

Description

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

Linters

There are not currently any linters tagged with 'deprecated'.

See Also

linters for a complete list of linters available in lintr.

---

download_file_linter *Recommend usage of a portable* mode *value for downloading files*

Description

`mode = "w"` (the default) or `mode = "a"` in `download.file()` can generate broken files on Windows. Instead, `utils::download.file()` recommends the usage of `mode = "wb"` and `mode = "ab"`. If `method = "curl"` or `method = "wget"`, no `mode` should be provided as it will be ignored.

Usage

```
download_file_linter()
```

Tags

best_practices, common_mistakes, robustness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "download.file(x = my_url)",
  linters = download_file_linter()
)

lint(
  text = "download.file(x = my_url, mode = 'w')",
  linters = download_file_linter()
)

lint(
  text = "download.file(x = my_url, method = 'curl', mode = 'wb')",
  linters = download_file_linter()
)

# okay
lint(
  text = "download.file(x = my_url, mode = 'wb')",
  linters = download_file_linter()
)

lint(
  text = "download.file(x = my_url, method = 'curl')",
  linters = download_file_linter()
)
```

---

duplicate_argument_linter

Duplicate argument linter

Description

Check for duplicate arguments in function calls. Some cases are run-time errors (e.g. `mean(x = 1:5, x = 2:3)`), otherwise this linter is used to discourage explicitly providing duplicate names to objects (e.g. `c(a = 1, a = 2)`). Duplicate-named objects are hard to work with programmatically and should typically be avoided.

Usage

```
duplicate_argument_linter(except = c("mutate", "transmute"))
```

Arguments

`except`	A character vector of function names as exceptions. Defaults to functions that allow sequential updates to variables, currently `dplyr::mutate()` and `dplyr::transmute()`.

Tags

common_mistakes, configurable, correctness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "list(x = 1, x = 2)",
  linters = duplicate_argument_linter()
)

lint(
  text = "fun(arg = 1, arg = 2)",
  linters = duplicate_argument_linter()
)

# okay
lint(
  text = "list(x = 1, x = 2)",
  linters = duplicate_argument_linter(except = "list")
)

lint(
  text = "df %>% dplyr::mutate(x = a + b, x = x + d)",
  linters = duplicate_argument_linter()
)
```

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
- boolean_arithmetic_linter
- consecutive_mutate_linter
- fixed_regex_linter
- if_switch_linter
- ifelse_censor_linter
- inner_combine_linter
- length_test_linter
- lengths_linter
- list2df_linter
- literal_coercion_linter
- matrix_apply_linter
- nested_ifelse_linter
- nrow_subset_linter
- nzchar_linter
- outer_negation_linter
- redundant_equals_linter
- redundant_ifelse_linter
- regex_subset_linter
- routine_registration_linter
- sample_int_linter
- scalar_in_linter
- seq_linter
- sort_linter
- string_boundary_linter
- unnecessary_concatenation_linter
- unnecessary_lambda_linter
- vector_logic_linter
- which_grepl_linter

See Also

linters for a complete list of linters available in lintr.

`empty_assignment_linter`
Block assignment of `{}`

Description

Assignment of `{}` is the same as assignment of `NULL`; use the latter for clarity. Closely related: `unnecessary_concatenation_linter()`.

Usage

```
empty_assignment_linter()
```

Tags

best_practices, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x <- {}",
  linters = empty_assignment_linter()
)

writeLines("x = {\n}")
lint(
  text = "x = {\n}",
  linters = empty_assignment_linter()
)

# okay
lint(
  text = "x <- { 3 + 4 }",
  linters = empty_assignment_linter()
)

lint(
  text = "x <- NULL",
  linters = empty_assignment_linter()
)
```

`equals_na_linter` *Equality check with NA linter*

Description

Check for `x == NA`, `x != NA` and `x %in% NA`. Such usage is almost surely incorrect – checks for missing values should be done with `is.na()`.

Usage

```
equals_na_linter()
```

Tags

common_mistakes, correctness, default, robustness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x == NA",
  linters = equals_na_linter()
)

lint(
  text = "x != NA",
  linters = equals_na_linter()
)

lint(
  text = "x %in% NA",
  linters = equals_na_linter()
)

# okay
lint(
  text = "is.na(x)",
  linters = equals_na_linter()
)

lint(
  text = "!is.na(x)",
  linters = equals_na_linter()
)
```

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. For package authors, note that this includes loading the package itself, e.g. with `pkgload::load_all()` or installing and attaching the package.

Linters

The following linters are tagged with 'executing':

- namespace_linter
- object_length_linter
- object_name_linter
- object_overwrite_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, pkg_testthat

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "expect_true(x > y)",
  linters = expect_comparison_linter()
)

lint(
  text = "expect_true(x <= y)",
  linters = expect_comparison_linter()
)

lint(
  text = "expect_true(x == (y == 2))",
  linters = expect_comparison_linter()
)

# okay
lint(
  text = "expect_gt(x, y)",
  linters = expect_comparison_linter()
)

lint(
  text = "expect_lte(x, y)",
  linters = expect_comparison_linter()
)

lint(
  text = "expect_identical(x, y == 2)",
  linters = expect_comparison_linter()
)

lint(
  text = "expect_true(x < y | x > y^2)",
  linters = expect_comparison_linter()
)
```

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

Description

This linter enforces the usage of `testthat::expect_identical()` as the default expectation for comparisons in a testthat suite. `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 which might set arguments such as `ignore_attr` or `tolerance`)

Tags

package_development, pkg_testthat

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "expect_equal(x, y)",
  linters = expect_identical_linter()
)

lint(
  text = "expect_true(identical(x, y))",
  linters = expect_identical_linter()
)

# okay
lint(
  text = "expect_identical(x, y)",
  linters = expect_identical_linter()
)
```

```
lint(
  text = "expect_equal(x, y, check.attributes = FALSE)",
  linters = expect_identical_linter()
)

lint(
  text = "expect_equal(x, y, tolerance = 1e-6)",
  linters = expect_identical_linter()
)
```

`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, pkg_testthat, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "expect_equal(length(x), 2L)",
  linters = expect_length_linter()
)

# okay
lint(
  text = "expect_length(x, 2L)",
  linters = expect_length_linter()
)
```

`expect_lint` *Lint expectation*

Description

These are expectation functions to test specified linters on sample code in the `testthat` testing framework.

- `expect_lint` asserts that specified lints are generated.
- `expect_no_lint` asserts that no lints are generated.

Usage

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

expect_no_lint(content, ..., 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 localization of base R error messages. This makes testing them reproducible on all systems irrespective of their native R language setting.
`ignore_order`	Logical, default `FALSE`. If `TRUE`, the order of the `checks` does not matter, e.g. lints with higher line numbers can come before those with lower line numbers, and the order of linters affecting the same line is also irrelevant.

Value

NULL, invisibly.

Examples

```
# no expected lint
expect_no_lint("a", trailing_blank_lines_linter())

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

# several expected lints
expect_lint("a\n\n", list("trailing blank", "trailing blank"), trailing_blank_lines_linter())
expect_lint(
  "a\n\n",
  list(
    list(message = "trailing blank", line_number = 2),
    list(message = "trailing blank", 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, pkg_testthat, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = 'expect_equal(names(x), "a")',
  linters = expect_named_linter()
)

# okay
lint(
  text = 'expect_named(x, "a")',
  linters = expect_named_linter()
)

lint(
  text = 'expect_equal(colnames(x), "a")',
  linters = expect_named_linter()
)

lint(
  text = 'expect_equal(dimnames(x), "a")',
  linters = expect_named_linter()
)
```

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

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, pkg_testthat, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "expect_true(!x)",
  linters = expect_not_linter()
)

# okay
lint(
  text = "expect_false(x)",
  linters = expect_not_linter()
)
```

expect_null_linter *Require usage of* `expect_null` *for checking* `NULL`

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, pkg_testthat

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "expect_equal(x, NULL)",
  linters = expect_null_linter()
)

lint(
  text = "expect_identical(x, NULL)",
  linters = expect_null_linter()
)

lint(
  text = "expect_true(is.null(x))",
  linters = expect_null_linter()
)


# okay
lint(
  text = "expect_null(x)",
  linters = expect_null_linter()
)
```

---

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, pkg_testthat

See Also

- linters for a complete list of linters available in lintr.
- `expect_s4_class_linter()`

Examples

```
# will produce lints
lint(
  text = 'expect_equal(class(x), "data.frame")',
  linters = expect_s3_class_linter()
)

lint(
  text = 'expect_equal(class(x), "numeric")',
  linters = expect_s3_class_linter()
)

# okay
lint(
  text = 'expect_s3_class(x, "data.frame")',
  linters = expect_s3_class_linter()
)

lint(
  text = 'expect_type(x, "double")',
  linters = expect_s3_class_linter()
)
```

`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, pkg_testthat

See Also

- linters for a complete list of linters available in lintr.
- `expect_s3_class_linter()`

Examples

```
# will produce lints
lint(
  text = 'expect_true(is(x, "Matrix"))',
  linters = expect_s4_class_linter()
)

# okay
lint(
  text = 'expect_s4_class(x, "Matrix")',
  linters = expect_s4_class_linter()
)
```

---

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, pkg_testthat, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "expect_equal(x, TRUE)",
  linters = expect_true_false_linter()
)

lint(
  text = "expect_equal(x, FALSE)",
  linters = expect_true_false_linter()
)

# okay
lint(
  text = "expect_true(x)",
  linters = expect_true_false_linter()
)

lint(
  text = "expect_false(x)",
  linters = expect_true_false_linter()
)
```

---

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, pkg_testthat

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = 'expect_equal(typeof(x), "double")',
  linters = expect_type_linter()
)

lint(
  text = 'expect_identical(typeof(x), "double")',
  linters = expect_type_linter()
)

# okay
lint(
  text = 'expect_type(x, "double")',
  linters = expect_type_linter()
)
```

`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(allow_unescaped = FALSE)
```

Arguments

`allow_unescaped`
: Logical, default `FALSE`. If `TRUE`, only patterns that require regex escapes (e.g. `"\\$"` or `"[$]"`) will be linted. See examples.

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, configurable, efficiency, readability, regex

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
code_lines <- 'gsub("\\\\.", "", x)'
writeLines(code_lines)
lint(
  text = code_lines,
  linters = fixed_regex_linter()
)

lint(
  text = 'grepl("a[*]b", x)',
  linters = fixed_regex_linter()
)

lint(
  text = 'grepl("a[*]b", x)',
  linters = fixed_regex_linter(allow_unescaped = TRUE)
)

code_lines <- 'stringr::str_subset(x, "\\\\$")'
writeLines(code_lines)
lint(
  text = code_lines,
  linters = fixed_regex_linter()
)

lint(
  text = 'grepl("Munich", address)',
  linters = fixed_regex_linter()
)

# okay
code_lines <- 'gsub("\\\\.", "", x, fixed = TRUE)'
writeLines(code_lines)
lint(
  text = code_lines,
  linters = fixed_regex_linter()
)

lint(
  text = 'grepl("a*b", x, fixed = TRUE)',
  linters = fixed_regex_linter()
)

lint(
  text = 'stringr::str_subset(x, stringr::fixed("$"))',
  linters = fixed_regex_linter()
)
```

```
lint(
  text = 'grepl("Munich", address, fixed = TRUE)',
  linters = fixed_regex_linter()
)

lint(
  text = 'grepl("Munich", address)',
  linters = fixed_regex_linter(allow_unescaped = TRUE)
)
```

`for_loop_index_linter` *Block usage of for loops directly overwriting the indexing variable*

Description

`for (x in x)` is a poor choice of indexing variable. This overwrites `x` in the calling scope and is confusing to read.

Usage

```
for_loop_index_linter()
```

Tags

best_practices, readability, robustness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "for (x in x) { TRUE }",
  linters = for_loop_index_linter()
)

lint(
  text = "for (x in foo(x, y)) { TRUE }",
  linters = for_loop_index_linter()
)

# okay
lint(
  text = "for (xi in x) { TRUE }",
  linters = for_loop_index_linter()
)
```

```
lint(
  text = "for (col in DF$col) { TRUE }",
  linters = for_loop_index_linter()
)
```

`function_argument_linter` *Function argument linter*

Description

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

Changing the argument order can be a breaking change. An alternative to changing the argument order is to instead set the default for such arguments to `NULL`.

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/required-no-defaults.html

Examples

```
# will produce lints
lint(
  text = "function(y = 1, z = 2, x) {}",
  linters = function_argument_linter()
)

lint(
  text = "function(x, y, z = 1, ..., w) {}",
  linters = function_argument_linter()
)

# okay
lint(
  text = "function(x, y = 1, z = 2) {}",
  linters = function_argument_linter()
)
```

```
lint(
  text = "function(x, y, w, z = 1, ...) {}",
  linters = function_argument_linter()
)

lint(
  text = "function(y = 1, z = 2, x = NULL) {}",
  linters = function_argument_linter()
)

lint(
  text = "function(x, y, z = 1, ..., w = NULL) {}",
  linters = function_argument_linter()
)
```

`function_left_parentheses_linter`
Function left parentheses linter

Description

Check that all left parentheses in a function call do not have spaces before them (e.g. `mean (1:3)`). Although this is syntactically valid, it makes the code difficult to read.

Usage

```
function_left_parentheses_linter()
```

Details

Exceptions are made for control flow functions (`if`, `for`, etc.).

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()

Examples

```
# will produce lints
lint(
  text = "mean (x)",
  linters = function_left_parentheses_linter()
)

lint(
  text = "stats::sd(c (x, y, z))",
  linters = function_left_parentheses_linter()
)

# okay
lint(
  text = "mean(x)",
  linters = function_left_parentheses_linter()
)

lint(
  text = "stats::sd(c(x, y, z))",
  linters = function_left_parentheses_linter()
)

lint(
  text = "foo <- function(x) (x + 1)",
  linters = function_left_parentheses_linter()
)
```

---

function_return_linter

Lint common mistakes/style issues cropping up from return statements

---

Description

`return(x <- ...)` is either distracting (because `x` is ignored), or confusing (because assigning to `x` has some side effect that is muddled by the dual-purpose expression).

Usage

```
function_return_linter()
```

Tags

best_practices, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "foo <- function(x) return(y <- x + 1)",
  linters = function_return_linter()
)

lint(
  text = "foo <- function(x) return(x <<- x + 1)",
  linters = function_return_linter()
)

writeLines("e <- new.env() \nfoo <- function(x) return(e$val <- x + 1)")
lint(
  text = "e <- new.env() \nfoo <- function(x) return(e$val <- x + 1)",
  linters = function_return_linter()
)

# okay
lint(
  text = "foo <- function(x) return(x + 1)",
  linters = function_return_linter()
)

code_lines <- "
foo <- function(x) {
  x <<- x + 1
  return(x)
}
"
lint(
  text = code_lines,
  linters = function_return_linter()
)

code_lines <- "
e <- new.env()
foo <- function(x) {
  e$val <- x + 1
  return(e$val)
}
"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = function_return_linter()
)
```

`get_r_string` *Extract text from* `STR_CONST` *nodes*

Description

Convert STR_CONST text() values into R strings. This is useful to account for arbitrary character literals, e.g. R"------[hello]------", which is parsed in R as "hello". It is quite cumbersome to write XPaths allowing for strings like this, so whenever your linter logic requires testing a STR_CONST node's value, use this function. NB: this is also properly vectorized on s, and accepts a variety of inputs. Empty inputs will become NA outputs, which helps ensure that length(get_r_string(s)) == length(s).

Usage

```
get_r_string(s, xpath = NULL)
```

Arguments

s	An input string or strings. If s is an xml_node or xml_nodeset and xpath is NULL, extract its string value with xml2::xml_text(). If s is an xml_node or xml_nodeset and xpath is specified, it is extracted with xml2::xml_find_chr().
xpath	An XPath, passed on to xml2::xml_find_chr() after wrapping with string().

Examples

```
tmp <- tempfile()
writeLines("c('a', 'b')", tmp)
expr_as_xml <- get_source_expressions(tmp)$expressions[[1L]]$xml_parsed_content
writeLines(as.character(expr_as_xml))
get_r_string(expr_as_xml, "expr[2]")
get_r_string(expr_as_xml, "expr[3]")
unlink(tmp)

# more importantly, extract raw strings correctly
tmp_raw <- tempfile()
writeLines("c(R'(a\\b)', R'--[a\\\"\'\"\\b]--')", tmp_raw)
expr_as_xml_raw <- get_source_expressions(tmp_raw)$expressions[[1L]]$xml_parsed_content
writeLines(as.character(expr_as_xml_raw))
get_r_string(expr_as_xml_raw, "expr[2]")
get_r_string(expr_as_xml_raw, "expr[3]")
unlink(tmp_raw)
```

---

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 using the encoding setting. This setting is 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 8 elements:

- filename (character) the name of the file.
- line (integer) the line in the file where this expression begins.
- column (integer) the column in the file where this expression begins.
- lines (named character) vector of all lines spanned by this expression, named with the corresponding line numbers.
- 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).
- xml_find_function_calls(function_names) (function) a function that returns all SYMBOL_FUNCTION_CALL XML nodes from xml_parsed_content with specified function names.

The final element of expressions is a list corresponding to the full file consisting of 7 elements:

- filename (character) the name of this file.
- file_lines (character) the readLines() output for this file.
- content (character) for .R files, the same as file_lines; for .Rmd or .qmd 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).
- `xml_find_function_calls(function_names)` (`function`) a function that returns all `SYMBOL_FUNCTION_CALL` XML nodes from `full_xml_parsed_content` with specified function names.

error A `Lint` object describing any parsing error.

warning A `lints` object describing any parsing warning.

lines The `readLines()` output for this file.

Examples

```
tmp <- tempfile()
writeLines(c("x <- 1", "y <- x + 1"), tmp)
get_source_expressions(tmp)
unlink(tmp)
```

`gitlab_output` *GitLab Report for lint results*

Description

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

Usage

```
gitlab_output(lints, filename = "lintr_results.json")
```

Arguments

`lints`	The linting results
`filename`	The file name of the output report

Details

lintr only supports three severity types ("style", "warning", and "error") while the GitLab format supports five ("info", "minor", "major", "critical", and "blocker"). The types "style", "warning", and "error" are mapped to the GitLab types "info", "major", and "blocker", respectively. The GitLab types "minor" and "critical" are ignored.

`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 = `==`)

with_id(source_expression, id)
```

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%`.
`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.

Examples

```
tmp <- tempfile()
writeLines(c("x <- 1", "y <- x + 1"), tmp)
source_exprs <- get_source_expressions(tmp)
ids_with_token(source_exprs$expressions[[1L]], value = "SYMBOL")
with_id(source_exprs$expressions[[1L]], 2L)
unlink(tmp)
```

---

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.

Examples

```
# will produce lints
lint(
  text = "ifelse(5:1 < pi, 5:1, pi)",
  linters = ifelse_censor_linter()
)

lint(
  text = "ifelse(x > 0, x, 0)",
  linters = ifelse_censor_linter()
)

# okay
```

```
lint(
  text = "pmin(5:1, pi)",
  linters = ifelse_censor_linter()
)

lint(
  text = "pmax(x, 0)",
  linters = ifelse_censor_linter()
)
```

`if_not_else_linter` *Block statements like if (!A) x else y*

Description

`if (!A) x else y` is the same as `if (A) y else x`, but the latter is easier to reason about in the `else` case. The former requires double negation that can be avoided by switching the statement order.

Usage

```
if_not_else_linter(exceptions = c("is.null", "is.na", "missing"))
```

Arguments

`exceptions`	Character vector of calls to exclude from linting. By default, `is.null()`, `is.na()`, and `missing()` are excluded given the common idiom `!is.na(x)` as "x is present".

Details

This only applies in the simple `if/else` case. Statements like `if (!A) x else if (B) y else z` don't always have a simpler or more readable form.

It also applies to `ifelse()` and the package equivalents `dplyr::if_else()` and `data.table::fifelse()`.

Tags

configurable, consistency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "if (!A) x else y",
  linters = if_not_else_linter()
)

lint(
  text = "if (!A) x else if (!B) y else z",
  linters = if_not_else_linter()
)

lint(
  text = "ifelse(!is_treatment, x, y)",
  linters = if_not_else_linter()
)

lint(
  text = "if (!is.null(x)) x else 2",
  linters = if_not_else_linter(exceptions = character())
)

# okay
lint(
  text = "if (A) x else y",
  linters = if_not_else_linter()
)

lint(
  text = "if (!A) x else if (B) z else y",
  linters = if_not_else_linter()
)

lint(
  text = "ifelse(is_treatment, y, x)",
  linters = if_not_else_linter()
)

lint(
  text = "if (!is.null(x)) x else 2",
  linters = if_not_else_linter()
)
```

`if_switch_linter` *Require usage of switch() over repeated if/else blocks*

Description

`switch()` statements in R are used to delegate behavior based on the value of some input scalar

string, e.g. `switch(x, a = 1, b = 3, c = 7, d = 8)` will be one of `1`, `3`, `7`, or `8`, depending on the value of `x`.

Usage

```
if_switch_linter(max_branch_lines = 0L, max_branch_expressions = 0L)
```

Arguments

`max_branch_lines`, `max_branch_expressions`
: Integer, default 0 indicates "no maximum". If set any `if/else if/.../else` chain where any branch occupies more than this number of lines (resp. expressions) will not be linted. The conjugate applies to `switch()` statements – if these parameters are set, any `switch()` statement with any overly-complicated branches will be linted. See examples.

Details

This can also be accomplished by repeated `if/else` statements like so: `if (x == "a") 1 else if (x == "b") 2 else if (x == "c") 7 else 8` (implicitly, the last `else` assumes `x` only takes 4 possible values), but this is more cluttered and slower (note that `switch()` takes the same time to evaluate regardless of the value of `x`, and is faster even when `x` takes the first value (here `a`), and that the `if/else` approach is roughly linear in the number of conditions that need to be evaluated, here up to 3 times).

Tags

best_practices, configurable, consistency, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "if (x == 'a') 1 else if (x == 'b') 2 else 3",
  linters = if_switch_linter()
)

code <- paste(
  "if (x == 'a') {",
  "  1",
  "} else if (x == 'b') {",
  "  2",
  "} else if (x == 'c') {",
  "  y <- x",
  "  z <- sqrt(match(y, letters))",
  "  z",
  "}",
```

```
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = if_switch_linter()
)

code <- paste(
  "if (x == 'a') {",
  "  1",
  "} else if (x == 'b') {",
  "  2",
  "} else if (x == 'c') {",
  "  y <- x",
  "  z <- sqrt(",
  "    match(y, letters)",
  "  )",
  "  z",
  "}",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = if_switch_linter()
)

code <- paste(
  "switch(x,",
  "  a = {",
  "    1",
  "    2",
  "    3",
  "  },",
  "  b = {",
  "    1",
  "    2",
  "  }",
  ")",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = if_switch_linter(max_branch_lines = 2L)
)

# okay
lint(
  text = "switch(x, a = 1, b = 2, 3)",
  linters = if_switch_linter()
)
```

```
# switch() version not as clear
lint(
  text = "if (x == 'a') 1 else if (x == 'b' & y == 2) 2 else 3",
  linters = if_switch_linter()
)

code <- paste(
  "if (x == 'a') {",
  "  1",
  "} else if (x == 'b') {",
  "  2",
  "} else if (x == 'c') {",
  "  y <- x",
  "  z <- sqrt(match(y, letters))",
  "  z",
  "}",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = if_switch_linter(max_branch_lines = 2L)
)

code <- paste(
  "if (x == 'a') {",
  "  1",
  "} else if (x == 'b') {",
  "  2",
  "} else if (x == 'c') {",
  "  y <- x",
  "  z <- sqrt(",
  "    match(y, letters)",
  "  )",
  "  z",
  "}",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = if_switch_linter(max_branch_expressions = 2L)
)

code <- paste(
  "switch(x,",
  "  a = {",
  "    1",
  "    2",
  "    3",
  "  },",
  "  b = {",
```

```
  "    1",
  "    2",
  "  }",
  ")",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = if_switch_linter(max_branch_lines = 3L)
)
```

`implicit_assignment_linter` *Avoid implicit assignment in function calls*

Description

Assigning inside function calls makes the code difficult to read, and should be avoided, except for functions that capture side-effects (e.g. `utils::capture.output()`).

Usage

```
implicit_assignment_linter(
  except = c("bquote", "expression", "expr", "quo", "quos", "quote"),
  allow_lazy = FALSE,
  allow_scoped = FALSE,
  allow_paren_print = FALSE
)
```

Arguments

`except`	A character vector of functions to be excluded from linting.
`allow_lazy`	logical, default `FALSE`. If `TRUE`, assignments that only trigger conditionally (e.g. in the RHS of `&&` or `||` expressions) are skipped.
`allow_scoped`	Logical, default `FALSE`. If `TRUE`, "scoped assignments", where the object is assigned in the statement beginning a branch and used only within that branch, are skipped.
`allow_paren_print`	Logical, default `FALSE`. If `TRUE`, assignments using `(` for auto-printing at the top-level are not linted.

Tags

best_practices, configurable, readability, style

See Also

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

Examples

```
# will produce lints
lint(
  text = "if (x <- 1L) TRUE",
  linters = implicit_assignment_linter()
)

lint(
  text = "mean(x <- 1:4)",
  linters = implicit_assignment_linter()
)

lint(
  text = "(x <- 1)",
  linters = implicit_assignment_linter()
)


# okay
lines <- "x <- 1L\nif (x) TRUE"
writeLines(lines)
lint(
  text = lines,
  linters = implicit_assignment_linter()
)

lines <- "x <- 1:4\nmean(x)"
writeLines(lines)
lint(
  text = lines,
  linters = implicit_assignment_linter()
)

lint(
  text = "A && (B <- foo(A))",
  linters = implicit_assignment_linter(allow_lazy = TRUE)
)

lines <- c(
  "if (any(idx <- x < 0)) {",
  "  stop('negative elements: ', toString(which(idx)))",
  "}"
)
writeLines(lines)
lint(
  text = lines,
  linters = implicit_assignment_linter(allow_scoped = TRUE)
```

```
)

lint(
  text = "(x <- 1)",
  linters = implicit_assignment_linter(allow_paren_print = TRUE)
)
```

`implicit_integer_linter` *Implicit integer linter*

Description

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

Usage

```
implicit_integer_linter(allow_colon = FALSE)
```

Arguments

`allow_colon`	Logical, default `FALSE`. If `TRUE`, expressions involving `:` won't throw a lint regardless of whether the inputs are implicitly integers.

Tags

best_practices, configurable, consistency, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x <- 1",
  linters = implicit_integer_linter()
)

lint(
  text = "x[2]",
  linters = implicit_integer_linter()
)

lint(
  text = "1:10",
  linters = implicit_integer_linter()
```

```
)

# okay
lint(
  text = "x <- 1.0",
  linters = implicit_integer_linter()
)

lint(
  text = "x <- 1L",
  linters = implicit_integer_linter()
)

lint(
  text = "x[2L]",
  linters = implicit_integer_linter()
)

lint(
  text = "1:10",
  linters = implicit_integer_linter(allow_colon = TRUE)
)
```

`indentation_linter` *Check that indentation is consistent*

Description

Check that indentation is consistent

Usage

```
indentation_linter(
  indent = 2L,
  hanging_indent_style = c("tidy", "always", "never"),
  assignment_as_infix = TRUE
)
```

Arguments

`indent`	Number of spaces, that a code block should be indented by relative to its parent code block. Used for multi-line code blocks (`{ ... }`), function calls (`( ... )`) and extractions (`[ ... ]`, `[[ ... ]]`). Defaults to 2.
`hanging_indent_style`	Indentation style for multi-line function calls with arguments in their first line. Defaults to tidyverse style, i.e. a block indent is used if the function call terminates with `)` on a separate line and a hanging indent if not. Note that function multi-line function calls without arguments on their first line will always

be expected to have block-indented arguments. If `hanging_indent_style` is `"tidy"`, multi-line function definitions are expected to be double-indented if the first line of the function definition contains no arguments and the closing parenthesis is not on its own line.

```
# complies to any style
map(
  x,
  f,
  additional_arg = 42
)

# complies to "tidy" and "never"
map(x, f,
  additional_arg = 42
)

# complies to "always"
map(x, f,
    additional_arg = 42
)

# complies to "tidy" and "always"
map(x, f,
    additional_arg = 42)

# complies to "never"
map(x, f,
  additional_arg = 42)

# complies to "tidy"
function(
    a,
    b) {
  # body
}
```

`assignment_as_infix`

Treat `<-` as a regular (i.e. left-associative) infix operator? This means, that infix operators on the right hand side of an assignment do not trigger a second level of indentation:

```
# complies to any style
variable <- a %+%
  b %+%
  c

# complies to assignment_as_infix = TRUE
variable <-
  a %+%
```

```
  b %+%
  c

# complies to assignment_as_infix = FALSE
variable <-
  a %+%
    b %+%
    c
```

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/functions.html#long-lines-1

Examples

```
# will produce lints
code_lines <- "if (TRUE) {\n1 + 1\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = indentation_linter()
)

code_lines <- "if (TRUE) {\n    1 + 1\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = indentation_linter()
)

code_lines <- "map(x, f,\n  additional_arg = 42\n)"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = indentation_linter(hanging_indent_style = "always")
)

code_lines <- "map(x, f,\n    additional_arg = 42)"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = indentation_linter(hanging_indent_style = "never")
)

# okay
```

```
code_lines <- "map(x, f,\n  additional_arg = 42\n)"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = indentation_linter()
)

code_lines <- "if (TRUE) {\n    1 + 1\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = indentation_linter(indent = 4)
)
```

`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 exclude from consideration for linting. Default is to include the following "low-precedence" operators: `+`, `-`, `~`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `&`, `&&`, `|`, `||`, `<-`, `:=`, `<<-`, `->`, `->>`, `=`, `/`, `*`, and any infix operator (exclude infixes by passing `"%%"`). Note that `"="` here includes three different operators, from the parser's point of view. To lint only some of these, pass the corresponding parse tags (i.e., some of `"EQ_ASSIGN"`, `"EQ_SUB"`, and `"EQ_FORMALS"`; see `utils::getParseData()`).

`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

configurable, default, readability, style

See Also

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

Examples

```
# will produce lints
lint(
  text = "x<-1L",
  linters = infix_spaces_linter()
)

lint(
  text = "1:4 %>%sum()",
  linters = infix_spaces_linter()
)

# okay
lint(
  text = "x <- 1L",
  linters = infix_spaces_linter()
)

lint(
  text = "1:4 %>% sum()",
  linters = infix_spaces_linter()
)

code_lines <- "
ab     <- 1L
abcdef <- 2L
"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = infix_spaces_linter(allow_multiple_spaces = TRUE)
)

lint(
  text = "a||b",
  linters = infix_spaces_linter(exclude_operators = "||")
)

lint(
  text = "sum(1:10, na.rm=TRUE)",
  linters = infix_spaces_linter(exclude_operators = "EQ_SUB")
)
```

---

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))`. The same equivalence holds for 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()
```

Details

Note that `strptime()` has one idiosyncrasy to be aware of, namely that auto-detected `format=` is set by the first matching input, which means that a case like `c(as.POSIXct("2024-01-01"), as.POSIXct("2024-01-01 01:02:03"))` gives different results to `as.POSIXct(c("2024-01-01", "2024-01-01 01:02:03"))`. This false positive is rare; a workaround where possible is to use consistent formatting, i.e., `"2024-01-01 00:00:00"` in the example.

Tags

consistency, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "c(log10(x), log10(y), log10(z))",
  linters = inner_combine_linter()
)

# okay
lint(
  text = "log10(c(x, y, z))",
  linters = inner_combine_linter()
)

lint(
  text = "c(log(x, base = 10), log10(x, base = 2))",
  linters = inner_combine_linter()
)
```

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.

Examples

```
tmp <- tempfile()
writeLines(c("x <- 1", "y <- x + 1"), tmp)
source_exprs <- get_source_expressions(tmp)
is_lint_level(source_exprs$expressions[[1L]], level = "expression")
is_lint_level(source_exprs$expressions[[1L]], level = "file")
is_lint_level(source_exprs$expressions[[3L]], level = "expression")
is_lint_level(source_exprs$expressions[[3L]], level = "file")
unlink(tmp)
```

is_numeric_linter *Redirect* `is.numeric(x) || is.integer(x)` *to just use* `is.numeric(x)`

Description

`is.numeric()` returns `TRUE` when `typeof(x)` is `double` or `integer` – testing `is.numeric(x) || is.integer(x)` is thus redundant.

Usage

```
is_numeric_linter()
```

Details

NB: This linter plays well with `class_equals_linter()`, which can help avoid further `is.numeric()` equivalents like `any(class(x) == c("numeric", "integer"))`.

Tags

best_practices, consistency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "is.numeric(y) || is.integer(y)",
  linters = is_numeric_linter()
)

lint(
  text = 'class(z) %in% c("numeric", "integer")',
  linters = is_numeric_linter()
)

# okay
lint(
  text = "is.numeric(y) || is.factor(y)",
  linters = is_numeric_linter()
)

lint(
  text = 'class(z) %in% c("numeric", "integer", "factor")',
  linters = is_numeric_linter()
)
```

---

keyword_quote_linter *Block unnecessary quoting in calls*

Description

Any valid symbol can be used as a keyword argument to an R function call. Sometimes, it is necessary to quote (or backtick) an argument that is not an otherwise valid symbol (e.g. creating a vector whose names have spaces); besides this edge case, quoting should not be done.

Usage

```
keyword_quote_linter()
```

Details

The most common source of violation for this is creating named vectors, lists, or data.frame-alikes, but it can be observed in other calls as well.

Similar reasoning applies to extractions with `$` or `@`.

Tags

consistency, readability, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = 'data.frame("a" = 1)',
  linters = keyword_quote_linter()
)

lint(
  text = "data.frame(`a` = 1)",
  linters = keyword_quote_linter()
)

lint(
  text = 'my_list$"key"',
  linters = keyword_quote_linter()
)

lint(
  text = 's4obj@"key"',
  linters = keyword_quote_linter()
)

# okay
lint(
  text = "data.frame(`a b` = 1)",
  linters = keyword_quote_linter()
)

lint(
  text = "my_list$`a b`",
  linters = keyword_quote_linter()
)
```

`lengths_linter` *Require usage of* `lengths()` *where possible*

Description

`base::lengths()` is a function that was added to base R in version 3.2.0 to get the length of each element of a list. It is equivalent to `sapply(x, length)`, but faster and more readable.

Usage

```
lengths_linter()
```

Tags

best_practices, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "sapply(x, length)",
  linters = lengths_linter()
)

lint(
  text = "vapply(x, length, integer(1L))",
  linters = lengths_linter()
)

lint(
  text = "purrr::map_int(x, length)",
  linters = lengths_linter()
)

# okay
lint(
  text = "lengths(x)",
  linters = lengths_linter()
)
```

length_levels_linter *Require usage of nlevels over length(levels(.))*

Description

`length(levels(x))` is the same as `nlevels(x)`, but harder to read.

Usage

```
length_levels_linter()
```

Tags

best_practices, consistency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "length(levels(x))",
  linters = length_levels_linter()
)

# okay
lint(
  text = "length(c(levels(x), levels(y)))",
  linters = length_levels_linter()
)
```

length_test_linter *Check for a common mistake where a size check like 'length' is applied in the wrong place*

Description

Usage like `length(x == 0)` is a mistake. If you intended to check `x` is empty, use `length(x) == 0`. Other mistakes are possible, but running `length()` on the outcome of a logical comparison is never the best choice.

Usage

```
length_test_linter()
```

Details

The linter also checks for similar usage with `nrow()`, `ncol()`, `NROW()`, and `NCOL()`.

Tags

best_practices, consistency, robustness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "length(x == 0)",
  linters = length_test_linter()
)

lint(
  text = "nrow(x > 0) || ncol(x > 0)",
  linters = length_test_linter()
)

lint(
  text = "NROW(x == 1) && NCOL(y == 1)",
  linters = length_test_linter()
)

# okay
lint(
  text = "length(x) > 0",
  linters = length_test_linter()
)

lint(
  text = "nrow(x) > 0 || ncol(x) > 0",
  linters = length_test_linter()
)

lint(
  text = "NROW(x) == 1 && NCOL(y) == 1",
  linters = length_test_linter()
)
```

`library_call_linter` *Library call linter*

Description

This linter covers several rules related to `library()` calls:

Usage

```
library_call_linter(allow_preamble = TRUE)
```

Arguments

`allow_preamble`	Logical, default `TRUE`. If `FALSE`, no code is allowed to precede the first `library()` call, otherwise some setup code is allowed, but all `library()` calls must follow consecutively after the first one.

Details

- Enforce such calls to all be at the top of the script.
- Block usage of argument `character.only`, in particular for loading packages in a loop.
- Block consecutive calls to `suppressMessages(library(.))` in favor of using `backports::suppressMessages()` only once to suppress messages from all `library()` calls. Ditto `suppressPackageStartupMessages()`.

Tags

best_practices, configurable, readability, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints

code <- "library(dplyr)\nprint('test')\nlibrary(tidyr)"
writeLines(code)
lint(
  text = code,
  linters = library_call_linter()
)

lint(
  text = "library('dplyr', character.only = TRUE)",
  linters = library_call_linter()
)
```

```
code <- paste(
  "pkg <- c('dplyr', 'tibble')",
  "sapply(pkg, library, character.only = TRUE)",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = library_call_linter()
)

code <- "suppressMessages(library(dplyr))\nsuppressMessages(library(tidyr))"
writeLines(code)
lint(
  text = code,
  linters = library_call_linter()
)

# okay
code <- "library(dplyr)\nprint('test')"
writeLines(code)
lint(
  text = code,
  linters = library_call_linter()
)

code <- "# comment\nlibrary(dplyr)"
lint(
  text = code,
  linters = library_call_linter()
)

code <- paste(
  "foo <- function(pkg) {",
  "  sapply(pkg, library, character.only = TRUE)",
  "}",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = library_call_linter()
)

code <- "suppressMessages({\n  library(dplyr)\n  library(tidyr)\n})"
writeLines(code)
lint(
  text = code,
  linters = library_call_linter()
)
```

`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, ignore_string_bodies = FALSE)
```

Arguments

`length`	Maximum line length allowed. Default is `80L` (Hollerith limit).
`ignore_string_bodies`	Logical, default `FALSE`. If `TRUE`, the contents of string literals are ignored. The quotes themselves are included, so this mainly affects wide multiline strings, e.g. SQL queries.

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

Examples

```
# will produce lints
lint(
  text = strrep("x", 23L),
  linters = line_length_linter(length = 20L)
)

# the trailing ' is counted towards line length, so this still lints
lint(
  text = "'a long single-line string'",
  linters = line_length_linter(length = 15L, ignore_string_bodies = TRUE)
)

lines <- paste(
  "query <- '",
  "  SELECT *",
  "  FROM MyTable",
  "  WHERE profit > 0",
  "'",
  sep = "\n"
```

```
)
writeLines(lines)
lint(
  text = lines,
  linters = line_length_linter(length = 10L)
)

# okay
lint(
  text = strrep("x", 21L),
  linters = line_length_linter(length = 40L)
)

lines <- paste(
  "paste(",
  "  'a long',",
  "  'single-line',",
  "  'string'",
  ")",
  sep = "\n"
)
writeLines(lines)
lint(
  text = lines,
  linters = line_length_linter(length = 15L, ignore_string_bodies = TRUE)
)

lines <- paste(
  "query <- '",
  "  SELECT *",
  "  FROM MyTable",
  "  WHERE profit > 0",
  "'",
  sep = "\n"
)
writeLines(lines)
lint(
  text = lines,
  linters = line_length_linter(length = 10L, ignore_string_bodies = TRUE)
)
```

`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/`, `demo/`, and `exec/`.

Usage

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

lint_dir(
  path = ".",
  ...,
  relative_path = TRUE,
  exclusions = list("renv", "packrat"),
  pattern = "(?i)[.](r|rmd|qmd|rnw|rhtml|rrst|rtex|rtxt)$",
  parse_settings = TRUE,
  show_progress = NULL
)

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

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	When logical, toggle caching of lint results. If passed a character string, store the cache in this directory.
parse_settings	Logical. Whether to try and parse the settings. Otherwise, the default_settings() are used. TRUE by default when linting files, as opposed to text=.
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, .qmd, .Rnw, .Rhtml, .Rrst, .Rtex, .Rtxt allowing for lowercase r (.r, ...).
show_progress	Logical controlling whether to show linting progress with `cli::cli_progress_along()`. The default behavior is to show progress in `interactive()` sessions not running a testthat suite.

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

An object of class `c("lints", "list")`, each element of which is a `"list"` object.

Examples

```
# linting inline-code
lint("a = 123\n")
lint(text = "a = 123")

# linting a file
f <- tempfile()
writeLines("a=1", f)
lint(f)
unlink(f)

if (FALSE) {
  lint_dir()

  lint_dir(
    linters = list(semicolon_linter()),
    exclusions = list(
      "inst/doc/creating_linters.R" = 1,
      "inst/example/bad.R",
      "renv"
    )
  )
}
if (FALSE) {
  lint_package()

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

```
  )
}
```

`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
)
```

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.

Value

an object of class `c("lint", "list")`.

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

Description

Create a `linter` closure

Usage

```
Linter(
  fun,
  name = linter_auto_name(),
  linter_level = c(NA_character_, "file", "expression")
)
```

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.
`linter_level`	Which level of expression is the linter working with? `"expression"` means an individual expression in `xml_parsed_content`, while `"file"` means all expressions in the current file are available in `full_xml_parsed_content`. `NA` means the linter will be run with both, expression-level and file-level source expressions.

Value

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

`linters` *Available linters*

Description

A variety of linters are 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 (65 linters)
- common_mistakes (13 linters)
- configurable (44 linters)
- consistency (33 linters)
- correctness (7 linters)
- default (26 linters)
- efficiency (30 linters)
- executing (6 linters)
- package_development (14 linters)
- pkg_testthat (12 linters)
- readability (66 linters)
- regex (4 linters)
- robustness (19 linters)
- style (40 linters)
- tidy_design (1 linters)

Linters

The following linters exist:

- `absolute_path_linter` (tags: best_practices, configurable, robustness)
- `all_equal_linter` (tags: common_mistakes, robustness)
- `any_duplicated_linter` (tags: best_practices, efficiency)
- `any_is_na_linter` (tags: best_practices, efficiency)
- `assignment_linter` (tags: configurable, consistency, default, style)
- `backport_linter` (tags: configurable, package_development, robustness)
- `boolean_arithmetic_linter` (tags: best_practices, efficiency, readability)
- `brace_linter` (tags: configurable, default, readability, style)
- `class_equals_linter` (tags: best_practices, consistency, robustness)
- `coalesce_linter` (tags: best_practices, consistency, readability)
- `commas_linter` (tags: configurable, default, readability, style)
- `commented_code_linter` (tags: best_practices, default, readability, style)
- `comparison_negation_linter` (tags: consistency, readability)
- `condition_call_linter` (tags: best_practices, configurable, style, tidy_design)
- `condition_message_linter` (tags: best_practices, consistency)
- `conjunct_test_linter` (tags: best_practices, configurable, package_development, pkg_testthat, readability)

- consecutive_assertion_linter (tags: consistency, readability, style)
- consecutive_mutate_linter (tags: configurable, consistency, efficiency, readability)
- cyclocomp_linter (tags: best_practices, configurable, readability, style)
- download_file_linter (tags: best_practices, common_mistakes, robustness)
- duplicate_argument_linter (tags: common_mistakes, configurable, correctness)
- empty_assignment_linter (tags: best_practices, readability)
- equals_na_linter (tags: common_mistakes, correctness, default, robustness)
- expect_comparison_linter (tags: best_practices, package_development, pkg_testthat)
- expect_identical_linter (tags: package_development, pkg_testthat)
- expect_length_linter (tags: best_practices, package_development, pkg_testthat, readability)
- expect_named_linter (tags: best_practices, package_development, pkg_testthat, readability)
- expect_not_linter (tags: best_practices, package_development, pkg_testthat, readability)
- expect_null_linter (tags: best_practices, package_development, pkg_testthat)
- expect_s3_class_linter (tags: best_practices, package_development, pkg_testthat)
- expect_s4_class_linter (tags: best_practices, package_development, pkg_testthat)
- expect_true_false_linter (tags: best_practices, package_development, pkg_testthat, readability)
- expect_type_linter (tags: best_practices, package_development, pkg_testthat)
- fixed_regex_linter (tags: best_practices, configurable, efficiency, readability, regex)
- for_loop_index_linter (tags: best_practices, readability, robustness)
- function_argument_linter (tags: best_practices, consistency, style)
- function_left_parentheses_linter (tags: default, readability, style)
- function_return_linter (tags: best_practices, readability)
- if_not_else_linter (tags: configurable, consistency, readability)
- if_switch_linter (tags: best_practices, configurable, consistency, efficiency, readability)
- ifelse_censor_linter (tags: best_practices, efficiency)
- implicit_assignment_linter (tags: best_practices, configurable, readability, style)
- implicit_integer_linter (tags: best_practices, configurable, consistency, style)
- indentation_linter (tags: configurable, default, readability, style)
- infix_spaces_linter (tags: configurable, default, readability, style)
- inner_combine_linter (tags: consistency, efficiency, readability)
- is_numeric_linter (tags: best_practices, consistency, readability)
- keyword_quote_linter (tags: consistency, readability, style)
- length_levels_linter (tags: best_practices, consistency, readability)
- length_test_linter (tags: common_mistakes, efficiency)
- lengths_linter (tags: best_practices, efficiency, readability)

- library_call_linter (tags: best_practices, configurable, readability, style)
- line_length_linter (tags: configurable, default, readability, style)
- list2df_linter (tags: efficiency, readability)
- list_comparison_linter (tags: best_practices, common_mistakes)
- literal_coercion_linter (tags: best_practices, consistency, efficiency)
- matrix_apply_linter (tags: efficiency, readability)
- 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)
- nested_pipe_linter (tags: configurable, consistency, readability)
- nonportable_path_linter (tags: best_practices, configurable, robustness)
- nrow_subset_linter (tags: best_practices, consistency, efficiency)
- numeric_leading_zero_linter (tags: consistency, readability, style)
- nzchar_linter (tags: best_practices, consistency, efficiency)
- object_length_linter (tags: configurable, default, executing, readability, style)
- object_name_linter (tags: configurable, consistency, default, executing, style)
- object_overwrite_linter (tags: best_practices, configurable, executing, readability, robustness)
- object_usage_linter (tags: configurable, correctness, default, executing, readability, style)
- one_call_pipe_linter (tags: 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)
- paste_linter (tags: best_practices, configurable, consistency)
- pipe_call_linter (tags: readability, style)
- pipe_consistency_linter (tags: configurable, default, readability, style)
- pipe_continuation_linter (tags: default, readability, style)
- pipe_return_linter (tags: best_practices, common_mistakes)
- print_linter (tags: best_practices, consistency)
- quotes_linter (tags: configurable, consistency, default, readability, style)
- redundant_equals_linter (tags: best_practices, common_mistakes, efficiency, readability)
- redundant_ifelse_linter (tags: best_practices, configurable, consistency, efficiency)
- regex_subset_linter (tags: best_practices, efficiency, regex)
- rep_len_linter (tags: best_practices, consistency, readability)
- repeat_linter (tags: readability, style)
- return_linter (tags: configurable, default, style)

- routine_registration_linter (tags: best_practices, efficiency, robustness)
- sample_int_linter (tags: efficiency, readability, robustness)
- scalar_in_linter (tags: best_practices, configurable, consistency, efficiency, readability)
- semicolon_linter (tags: configurable, default, readability, style)
- seq_linter (tags: best_practices, consistency, default, efficiency, robustness)
- sort_linter (tags: best_practices, efficiency, readability)
- spaces_inside_linter (tags: default, readability, style)
- spaces_left_parentheses_linter (tags: default, readability, style)
- sprintf_linter (tags: common_mistakes, correctness)
- stopifnot_all_linter (tags: best_practices, readability)
- string_boundary_linter (tags: configurable, efficiency, readability, regex)
- 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)
- terminal_close_linter (tags: best_practices, robustness)
- todo_comment_linter (tags: configurable, style)
- trailing_blank_lines_linter (tags: default, style)
- trailing_whitespace_linter (tags: configurable, default, style)
- undesirable_function_linter (tags: best_practices, configurable, robustness, style)
- undesirable_operator_linter (tags: best_practices, configurable, robustness, style)
- unnecessary_concatenation_linter (tags: configurable, efficiency, readability, style)
- unnecessary_lambda_linter (tags: best_practices, configurable, efficiency, readability)
- unnecessary_nesting_linter (tags: best_practices, configurable, consistency, readability)
- unnecessary_placeholder_linter (tags: best_practices, readability)
- unreachable_code_linter (tags: best_practices, configurable, readability)
- unused_import_linter (tags: best_practices, common_mistakes, configurable, executing)
- vector_logic_linter (tags: best_practices, common_mistakes, default, efficiency)
- which_grepl_linter (tags: consistency, efficiency, readability, regex)
- whitespace_linter (tags: consistency, default, style)
- yoda_test_linter (tags: best_practices, package_development, pkg_testthat, 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)
```

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 list of linters to modify. Must be named.

See Also

- linters_with_tags for basing off tags attached to linters, possibly across multiple packages.
- all_linters for basing off all available linters in lintr.
- 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()`
f <- tempfile()
writeLines("my_slightly_long_variable_name <- 2.3", f)
lint(f, linters = linters_with_defaults(line_length_linter = line_length_linter(120L)))
unlink(f)

# the default linter list with a different line length cutoff
my_linters <- linters_with_defaults(line_length_linter = line_length_linter(120L))

# 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()
)
```

```
# checking the included linters
names(my_linters)
```

`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. See `available_tags("lintr")` to find out what tags are already used by lintr.
`...`	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 `exclude_tags` is `NULL`, no linters will be excluded. Note that `tags` takes priority, meaning that any tag found in both `tags` and `exclude_tags` will be included, not excluded. Note that linters with tag `"defunct"` (which do not work and can no longer be run) cannot be queried directly. See lintr-deprecated instead.

Value

A modified list of linters.

See Also

- linters_with_defaults for basing off lintr's set of default linters.
- all_linters for basing off all available linters in lintr.
- 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 <- linters_with_tags(tags = c("package_development", "style"))
names(linters)

# Get all linters tagged as "default" from lintr and mypkg
if (FALSE) {
  linters_with_tags("default", packages = c("lintr", "mypkg"))
}
```

---

list2df_linter *Recommend direct usage of* `data.frame()` *to create a data.frame from a list*

Description

`base::list2DF()` is the preferred way to turn a list of columns into a data.frame. Note that it doesn't support recycling; if that's required, use `data.frame()`.

Usage

```
list2df_linter()
```

Tags

efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "do.call(cbind.data.frame, x)",
  linters = list2df_linter()
)

lint(
  text = "do.call('cbind.data.frame', x)",
  linters = list2df_linter()
)

lint(
  text = "do.call(cbind.data.frame, list(a = 1, b = 1:10))",
```

```
  linters = list2df_linter()
)

# okay
lint(
  text = "list2df(x)",
  linters = list2df_linter()
)

lint(
  text = "data.frame(list(a = 1, b = 1:10))",
  linters = list2df_linter()
)
```

`list_comparison_linter` *Block usage of comparison operators with known-list() functions like lapply*

Description

Usage like `lapply(x, sum) > 10` is awkward because the list must first be coerced to a vector for comparison. A function like `vapply()` should be preferred.

Usage

```
list_comparison_linter()
```

Tags

best_practices, common_mistakes

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "lapply(x, sum) > 10",
  linters = list_comparison_linter()
)

# okay
lint(
  text = "unlist(lapply(x, sum)) > 10",
  linters = list_comparison_linter()
)
```

`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.

Examples

```
# will produce lints
lint(
  text = "int(1)",
  linters = literal_coercion_linter()
)

lint(
  text = "as.character(NA)",
  linters = literal_coercion_linter()
)

lint(
  text = "rlang::lgl(1L)",
  linters = literal_coercion_linter()
)

# okay
lint(
  text = "1L",
  linters = literal_coercion_linter()
```

```
)

lint(
  text = "NA_character_",
  linters = literal_coercion_linter()
)

lint(
  text = "TRUE",
  linters = literal_coercion_linter()
)
```

`make_linter_from_xpath` *Create a linter from an XPath*

Description

Create a linter from an XPath

Usage

```
make_linter_from_xpath(
  xpath,
  lint_message,
  type = c("warning", "style", "error"),
  level = c("expression", "file")
)

make_linter_from_function_xpath(
  function_names,
  xpath,
  lint_message,
  type = c("warning", "style", "error"),
  level = c("expression", "file")
)
```

Arguments

`xpath`	Character string, an XPath identifying R code to lint. For `make_linter_from_function_xpath()`, the XPath is relative to the `parent::expr` of the `SYMBOL_FUNCTION_CALL` nodes of the selected functions. See `xmlparsedata::xml_parse_data()` and `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.

`level`	Which level of expression is being tested? `"expression"` means an individual expression, while `"file"` means all expressions in the current file are available.
`function_names`	Character vector, names of functions whose calls to examine..

Examples

```
number_linter <- make_linter_from_xpath("//NUM_CONST", "This is a number.")
lint(text = "1 + 2", linters = number_linter())
```

---

matrix_apply_linter *Require usage of* `colSums(x)` *or* `rowSums(x)` *over* `apply(x, ., sum)`

Description

`colSums()` and `rowSums()` are clearer and more performant alternatives to `apply(x, 2, sum)` and `apply(x, 1, sum)` respectively in the case of 2D arrays, or matrices

Usage

```
matrix_apply_linter()
```

Tags

efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "apply(x, 1, sum)",
  linters = matrix_apply_linter()
)

lint(
  text = "apply(x, 2, sum)",
  linters = matrix_apply_linter()
)

lint(
  text = "apply(x, 2, sum, na.rm = TRUE)",
  linters = matrix_apply_linter()
)

lint(
  text = "apply(x, 2:4, sum)",
```

```
  linters = matrix_apply_linter()
)

# okay
lint(
  text = "rowSums(x)",
  linters = matrix_apply_linter()
)

lint(
  text = "colSums(x)",
  linters = matrix_apply_linter()
)

lint(
  text = "colSums(x, na.rm = TRUE)",
  linters = matrix_apply_linter()
)

lint(
  text = "rowSums(colSums(x), dims = 3)",
  linters = matrix_apply_linter()
)
```

`missing_argument_linter` *Missing argument linter*

Description

Check for missing arguments in function calls (e.g. `stats::median(1:10, )`).

Usage

```
missing_argument_linter(
  except = c("alist", "quote", "switch"),
  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.

Examples

```
# will produce lints
lint(
  text = 'tibble(x = "a", )',
  linters = missing_argument_linter()
)

# okay
lint(
  text = 'tibble(x = "a")',
  linters = missing_argument_linter()
)

lint(
  text = 'tibble(x = "a", )',
  linters = missing_argument_linter(except = "tibble")
)

lint(
  text = 'tibble(x = "a", )',
  linters = missing_argument_linter(allow_trailing = TRUE)
)
```

---

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.

Examples

```
# will produce lints
lint(
  text = "library(xyzxyz)",
  linters = missing_package_linter()
)

# okay
lint(
  text = "library(stats)",
  linters = missing_package_linter()
)
```

`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_defaults for basing off lintr's set of default linters.
- all_linters for basing off all available linters in lintr.
- 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

```
# custom list of undesirable functions:
#    remove `sapply` (using `NULL`)
#    add `cat` (with an accompanying message),
#    add `print` (unnamed, i.e. with no accompanying message)
#    add `source` (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[["source"]]
)

# list names of functions specified as undesirable
names(my_undesirable_functions)
```

---

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.

Examples

```
# will produce lints
lint(
  text = "xyzxyz::sd(c(1, 2, 3))",
  linters = namespace_linter()
)
```

```
lint(
  text = "stats::ssd(c(1, 2, 3))",
  linters = namespace_linter()
)

# okay
lint(
  text = "stats::sd(c(1, 2, 3))",
  linters = namespace_linter()
)

lint(
  text = "stats::ssd(c(1, 2, 3))",
  linters = namespace_linter(check_exports = FALSE)
)

lint(
  text = "stats:::ssd(c(1, 2, 3))",
  linters = namespace_linter(check_nonexports = FALSE)
)
```

`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/16275149); 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.

Let's say this is our original code:

```
ifelse(
  x == "a",
  2L,
  ifelse(x == "b", 3L, 1L)
)
```

Here are a few ways to avoid nesting and make the code more readable:

- Use `data.table::fcase()`

```
data.table::fcase(
  x == "a", 2L,
  x == "b", 3L,
  default = 1L
)
```

- Use `dplyr::case_match()`

```
dplyr::case_match(
  x,
  "a" ~ 2L,
  "b" ~ 3L,
  .default = 1L
)
```

- Use a look-up-and-merge approach (build a mapping table between values and outputs and merge this to the input)

```
default <- 1L
values <- data.frame(
  a = 2L,
  b = 3L
)
found_value <- values[[x]]
ifelse(is.null(found_value), default, found_value)
```

Tags

efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = 'ifelse(x == "a", 1L, ifelse(x == "b", 2L, 3L))',
  linters = nested_ifelse_linter()
)

# okay
lint(
  text = 'dplyr::case_when(x == "a" ~ 1L, x == "b" ~ 2L, TRUE ~ 3L)',
  linters = nested_ifelse_linter()
)
```

```
lint(
  text = 'data.table::fcase(x == "a", 1L, x == "b", 2L, default = 3L)',
  linters = nested_ifelse_linter()
)
```

`nested_pipe_linter` *Block usage of pipes nested inside other calls*

Description

Nesting pipes harms readability; extract sub-steps to separate variables, append further pipeline steps, or otherwise refactor such usage away.

Usage

```
nested_pipe_linter(
  allow_inline = TRUE,
  allow_outer_calls = c("try", "tryCatch", "withCallingHandlers")
)
```

Arguments

`allow_inline`	Logical, default `TRUE`, in which case only "inner" pipelines which span more than one line are linted. If `FALSE`, even "inner" pipelines that fit in one line are linted.
`allow_outer_calls`	Character vector dictating which "outer" calls to exempt from the requirement to unnest (see examples). Defaults to `try()`, `tryCatch()`, and `withCallingHandlers()`.

Tags

configurable, consistency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
code <- "df1 %>%\n  inner_join(df2 %>%\n    select(a, b)\n  )"
writeLines(code)
lint(
  text = code,
  linters = nested_pipe_linter()
)

lint(
```

```
  text = "df1 %>% inner_join(df2 %>% select(a, b))",
  linters = nested_pipe_linter(allow_inline = FALSE)
)

lint(
  text = "tryCatch(x %>% filter(grp == 'a'), error = identity)",
  linters = nested_pipe_linter(allow_outer_calls = character())
)

# okay
lint(
  text = "df1 %>% inner_join(df2 %>% select(a, b))",
  linters = nested_pipe_linter()
)

code <- "df1 %>%\n  inner_join(df2 %>%\n    select(a, b)\n  )"
writeLines(code)
lint(
  text = code,
  linters = nested_pipe_linter(allow_outer_calls = "inner_join")
)

lint(
  text = "tryCatch(x %>% filter(grp == 'a'), error = identity)",
  linters = nested_pipe_linter()
)
```

`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.
- absolute_path_linter()

Examples

```
# will produce lints
lint(
  text = "'abcdefg/hijklmnop/qrst/uv/wxyz'",
  linters = nonportable_path_linter()
)

# okay
lint(
  text = "file.path('abcdefg', 'hijklmnop', 'qrst', 'uv', 'wxyz')",
  linters = nonportable_path_linter()
)
```

---

nrow_subset_linter *Block usage of* nrow(subset(x, .))

Description

Using nrow(subset(x, condition)) to count the instances where condition applies inefficiently requires doing a full subset of x just to count the number of rows in the resulting subset. There are a number of equivalent expressions that don't require the full subset, e.g. with(x, sum(condition)) (or, more generically, with(x, sum(condition, na.rm = TRUE))).

Usage

```
nrow_subset_linter()
```

Tags

best_practices, consistency, efficiency

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "nrow(subset(x, is_treatment))",
  linters = nrow_subset_linter()
)

lint(
  text = "nrow(filter(x, is_treatment))",
  linters = nrow_subset_linter()
)

lint(
  text = "x %>% filter(x, is_treatment) %>% nrow()",
  linters = nrow_subset_linter()
)

# okay
lint(
  text = "with(x, sum(is_treatment, na.rm = TRUE))",
  linters = nrow_subset_linter()
)
```

---

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.

Examples

```
# will produce lints
lint(
  text = "x <- .1",
  linters = numeric_leading_zero_linter()
)

lint(
  text = "x <- -.1",
  linters = numeric_leading_zero_linter()
)

# okay
lint(
  text = "x <- 0.1",
  linters = numeric_leading_zero_linter()
)

lint(
  text = "x <- -0.1",
  linters = numeric_leading_zero_linter()
)
```

---

nzchar_linter *Require usage of nzchar where appropriate*

Description

`nzchar()` efficiently determines which of a vector of strings are empty (i.e., are `""`). It should in most cases be used instead of constructions like `string == ""` or `nchar(string) == 0`.

Usage

```
nzchar_linter()
```

Details

One crucial difference is in the default handling of `NA_character_`, i.e., missing strings. `nzchar(NA_character_)` is `TRUE`, while `NA_character_ == ""` and `nchar(NA_character_) == 0` are both `NA`. Therefore, for strict compatibility, use `nzchar(x, keepNA = TRUE)`. If the input is known to be complete (no missing entries), this argument can be dropped for conciseness.

Tags

best_practices, consistency, efficiency

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x[x == '']",
  linters = nzchar_linter()
)

lint(
  text = "x[nchar(x) > 0]",
  linters = nzchar_linter()
)

# okay
lint(
  text = "x[!nzchar(x, keepNA = TRUE)]",
  linters = nzchar_linter()
)

lint(
  text = "x[nzchar(x, keepNA = TRUE)]",
  linters = nzchar_linter()
)
```

---

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.

Examples

```
# will produce lints
lint(
  text = "very_very_long_variable_name <- 1L",
  linters = object_length_linter(length = 10L)
)

# okay
lint(
  text = "very_very_long_variable_name <- 1L",
  linters = object_length_linter(length = 30L)
)

lint(
  text = "var <- 1L",
  linters = object_length_linter(length = 10L)
)
```

---

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"), regexes = character())
```

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. The "symbols" style refers to names containing *only* non-alphanumeric characters; e.g., defining %+% from ggplot2 or %>% from magrittr would not generate lint markers, whereas %m+% from lubridate (containing both alphanumeric *and* non-alphanumeric characters) would.

`regexes` A (possibly named) character vector specifying a custom naming convention. If named, the names will be used in the lint message. Otherwise, the regexes enclosed by `/` will be used in the lint message. Note that specifying `regexes` overrides the default `styles`. So if you want to combine `regexes` and `styles`, both need to be explicitly specified.

Details

Quotes (`` `"' ``) and specials (`%` and trailing `<-`) are not considered part of the object name.

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.

Examples

```
# will produce lints
lint(
  text = "my_var <- 1L",
  linters = object_name_linter(styles = "CamelCase")
)

lint(
  text = "xYz <- 1L",
  linters = object_name_linter(styles = c("UPPERCASE", "lowercase"))
)

lint(
  text = "MyVar <- 1L",
  linters = object_name_linter(styles = "dotted.case")
)

lint(
  text = "asd <- 1L",
  linters = object_name_linter(regexes = c(my_style = "F$", "f$"))
```

```
)

# okay
lint(
  text = "my_var <- 1L",
  linters = object_name_linter(styles = "snake_case")
)

lint(
  text = "xyz <- 1L",
  linters = object_name_linter(styles = "lowercase")
)

lint(
  text = "my.var <- 1L; myvar <- 2L",
  linters = object_name_linter(styles = c("dotted.case", "lowercase"))
)

lint(
  text = "asdf <- 1L; asdF <- 1L",
  linters = object_name_linter(regexes = c(my_style = "F$", "f$"))
)
```

`object_overwrite_linter` *Block assigning any variables whose name clashes with a* `base` *R function*

Description

Re-using existing names creates a risk of subtle error best avoided. Avoiding this practice also encourages using better, more descriptive names.

Usage

```
object_overwrite_linter(
  packages = c("base", "stats", "utils", "tools", "methods", "graphics", "grDevices"),
  allow_names = character()
)
```

Arguments

`packages`	Character vector of packages to search for names that should be avoided. Defaults to the most common default packages: base, stats, utils, tools, methods, graphics, and grDevices.
`allow_names`	Character vector of object names to ignore, i.e., which are allowed to collide with exports from `packages`.

Tags

best_practices, configurable, executing, readability, robustness

See Also

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

Examples

```
# will produce lints
code <- "function(x) {\n  data <- x\n  data\n}"
writeLines(code)
lint(
  text = code,
  linters = object_overwrite_linter()
)

code <- "function(x) {\n  lint <- 'fun'\n  lint\n}"
writeLines(code)
lint(
  text = code,
  linters = object_overwrite_linter(packages = "lintr")
)

# okay
code <- "function(x) {\n  data('mtcars')\n}"
writeLines(code)
lint(
  text = code,
  linters = object_overwrite_linter()
)

code <- "function(x) {\n  data <- x\n  data\n}"
writeLines(code)
lint(
  text = code,
  linters = object_overwrite_linter(packages = "base")
)

# names in function signatures are ignored
lint(
  text = "function(data) data <- subset(data, x > 0)",
  linters = object_overwrite_linter()
)
```

`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 = NULL,
  interpret_extensions = c("glue", "rlang"),
  skip_with = TRUE
)
```

Arguments

`interpret_glue`	(Deprecated) If `TRUE`, interpret `glue::glue()` calls to avoid false positives caused by local variables which are only used in a glue expression. Provide `interpret_extensions` instead, see below.
`interpret_extensions`	Character vector of extensions to interpret. These are meant to cover known cases where variables may be used in ways understood by the reader but not by `checkUsage()` to avoid false positives. Currently `"glue"` and `"rlang"` are supported, both of which are in the default. • For `glue`, examine `glue::glue()` calls. • For `rlang`, examine `.env$key` usages.
`skip_with`	A logical. If `TRUE` (default), code in `with()` expressions will be skipped. This argument will be passed to `skipWith` argument of `codetools::checkUsage()`.

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.

Examples

```
# will produce lints
lint(
  text = "foo <- function() { x <- 1 }",
  linters = object_usage_linter()
)

# okay
lint(
  text = "foo <- function(x) { x <- 1 }",
  linters = object_usage_linter()
)

lint(
  text = "foo <- function() { x <- 1; return(x) }",
  linters = object_usage_linter()
)
```

---

one_call_pipe_linter *Block single-call magrittr pipes*

Description

Prefer using a plain call instead of a pipe with only one call, i.e. `1:10 %>% sum()` should instead be `sum(1:10)`. Note that calls in the first `%>%` argument count. `rowSums(x) %>% max()` is OK because there are two total calls (`rowSums()` and `max()`).

Usage

```
one_call_pipe_linter()
```

Details

Note also that un-"called" steps are *not* counted, since they should be calls (see `pipe_call_linter()`).

Tags

readability, style

See Also

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

Examples

```
# will produce lints
lint(
  text = "(1:10) %>% sum()",
  linters = one_call_pipe_linter()
)

lint(
  text = "DT %>% .[grp == 'a', sum(v)]",
  linters = one_call_pipe_linter()
)

# okay
lint(
  text = "rowSums(x) %>% mean()",
  linters = one_call_pipe_linter()
)

lint(
  text = "DT[src == 'a', .N, by = grp] %>% .[N > 10]",
  linters = one_call_pipe_linter()
)

# assignment pipe is exempted
lint(
  text = "DF %<>% mutate(a = 2)",
  linters = one_call_pipe_linter()
)
```

---

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

Description

`any(!x)` is logically equivalent to `!all(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.

Examples

```
# will produce lints
lint(
  text = "all(!x)",
  linters = outer_negation_linter()
)

lint(
  text = "any(!x)",
  linters = outer_negation_linter()
)

# okay
lint(
  text = "!any(x)",
  linters = outer_negation_linter()
)

lint(
  text = "!all(x)",
  linters = outer_negation_linter()
)
```

---

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()`, `.onDetach()`, and `.onUnload()` 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()`, `.onDetach()`, and `.onUnload()` 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.

Examples

```
# will produce lints
lint(
  text = ".onLoad <- function(lib, ...) { }",
  linters = package_hooks_linter()
)

lint(
  text = ".onAttach <- function(lib, pkg) { require(foo) }",
  linters = package_hooks_linter()
)

lint(
  text = ".onDetach <- function(pkg) { }",
  linters = package_hooks_linter()
)

lint(
  text = ".onUnload <- function() { }",
  linters = package_hooks_linter()
)

# okay
lint(
  text = ".onLoad <- function(lib, pkg) { }",
  linters = package_hooks_linter()
)

lint(
  text = '.onAttach <- function(lib, pkg) { loadNamespace("foo") }',
  linters = package_hooks_linter()
)

lint(
  text = ".onDetach <- function(lib) { }",
  linters = package_hooks_linter()
)

lint(
  text = ".onUnload <- function(libpath) { }",
  linters = package_hooks_linter()
)
```

---

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

Examples

```
# will produce lints
lint(
  text = "function(x)x + 1",
  linters = paren_body_linter()
)

# okay
lint(
  text = "function(x) x + 1",
  linters = paren_body_linter()
)
```

---

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

Description

The following issues are linted by default by this linter (see arguments for which can be de-activated optionally):

1. Block usage of base::paste() with sep = "". base::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.
4. Block usage of paste() / paste0() combined with rep() that could be replaced by base::strrep(). strrep() can handle the task of building a block of repeated strings (e.g. often used to build "horizontal lines" for messages). This is both more readable and skips the (likely small) overhead of putting two strings into the global string cache when only one is needed.

   Only target scalar usages – strrep can handle more complicated cases (e.g. strrep(letters, 26:1), but those aren't as easily translated from a paste(collapse=) call.

Usage

```
paste_linter(
  allow_empty_sep = FALSE,
  allow_to_string = FALSE,
  allow_file_path = c("double_slash", "always", "never")
)
```

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.

`allow_file_path`
: String, one of `"never"`, `"double_slash"`, or `"always"`; `"double_slash"` by default. If `"always"`, usage of `paste()` and `paste0()` to construct file paths is not linted. If `"double_slash"`, strings containing consecutive forward slashes will not lint. The main use case here is for URLs – "paths" like `"https://"` will not induce lints, since constructing them with `file.path()` might be deemed unnatural. Lastly, if `"never"`, strings with consecutive forward slashes will also lint. Note that `"//"` is never linted when it comes at the beginning or end of the input, to avoid requiring empty inputs like `file.path("", ...)` or `file.path(..., "")`.

Tags

best_practices, configurable, consistency

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = 'paste("a", "b", sep = "")',
  linters = paste_linter()
)

lint(
  text = 'paste(c("a", "b"), collapse = ", ")',
  linters = paste_linter()
)

lint(
  text = 'paste0(c("a", "b"), sep = " ")',
  linters = paste_linter()
)
```

```
lint(
  text = 'paste0(rep("*", 10L), collapse = "")',
  linters = paste_linter()
)

lint(
  text = 'paste0("http://site.com/", path)',
  linters = paste_linter(allow_file_path = "never")
)

lint(
  text = 'paste0(x, collapse = "")',
  linters = paste_linter()
)

# okay
lint(
  text = 'paste0("a", "b")',
  linters = paste_linter()
)

lint(
  text = 'paste("a", "b", sep = "")',
  linters = paste_linter(allow_empty_sep = TRUE)
)

lint(
  text = 'toString(c("a", "b"))',
  linters = paste_linter()
)

lint(
  text = 'paste(c("a", "b"), collapse = ", ")',
  linters = paste_linter(allow_to_string = TRUE)
)

lint(
  text = 'paste(c("a", "b"))',
  linters = paste_linter()
)

lint(
  text = 'strrep("*", 10L)',
  linters = paste_linter()
)

lint(
  text = 'paste0(year, "/", month, "/", day)',
  linters = paste_linter(allow_file_path = "always")
)

lint(
```

```
  text = 'paste0("http://site.com/", path)',
  linters = paste_linter()
)

lint(
  text = 'paste(x, collapse = "")',
  linters = paste_linter()
)
```

`pipe_call_linter` *Pipe call linter*

Description

Force explicit calls in magrittr pipes, e.g., `1:3 %>% sum()` instead of `1:3 %>% sum`. Note that native pipe always requires a function call, i.e. `1:3 |> sum` will produce an error.

Usage

```
pipe_call_linter()
```

Tags

readability, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "1:3 %>% mean %>% as.character",
  linters = pipe_call_linter()
)

# okay
lint(
  text = "1:3 %>% mean() %>% as.character()",
  linters = pipe_call_linter()
)
```

`pipe_consistency_linter` *Pipe consistency linter*

Description

Check that the recommended pipe operator is used, or more conservatively that pipes are consistent by file.

Usage

```
pipe_consistency_linter(pipe = c("|>", "%>%", "auto"))
```

Arguments

`pipe`	Which pipe operator is valid (either `"%>%"` or `"|>"`). The default is the native pipe (`|>`). `"auto"` will instead only enforce consistency, i.e., that in any given file there is only one pipe.

Tags

configurable, default, readability, style

See Also

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

Examples

```
# will produce lints
lint(
  text = "1:3 |> mean() %>% as.character()",
  linters = pipe_consistency_linter()
)

lint(
  text = "1:3 %>% mean() %>% as.character()",
  linters = pipe_consistency_linter("|>")
)

# okay
lint(
  text = "1:3 |> mean() |> as.character()",
  linters = pipe_consistency_linter()
)

lint(
  text = "1:3 %>% mean() %>% as.character()",
```

```
  linters = pipe_consistency_linter("%>%")
)
```

`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

Examples

```
# will produce lints
code_lines <- "1:3 %>%\n mean() %>% as.character()"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = pipe_continuation_linter()
)

code_lines <- "1:3 |> mean() |>\n as.character()"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = pipe_continuation_linter()
)

# okay
lint(
  text = "1:3 %>% mean() %>% as.character()",
  linters = pipe_continuation_linter()
)

code_lines <- "1:3 %>%\n mean() %>%\n as.character()"
writeLines(code_lines)
```

```
lint(
  text = code_lines,
  linters = pipe_continuation_linter()
)

lint(
  text = "1:3 |> mean() |> as.character()",
  linters = pipe_continuation_linter()
)

code_lines <- "1:3 |>\n mean() |>\n as.character()"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = pipe_continuation_linter()
)
```

`pipe_return_linter` *Block usage of return() in magrittr pipelines*

Description

`return()` inside a magrittr pipeline does not actually execute `return()` like you'd expect:

Usage

```
pipe_return_linter()
```

Details

```
bad_usage <- function(x) {
  x %>%
    return()
  FALSE
}
```

`bad_usage(TRUE)` will return `FALSE`! It will technically work "as expected" if this is the final statement in the function body, but such usage is misleading. Instead, assign the pipe outcome to a variable and return that.

Tags

best_practices, common_mistakes

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "function(x) x %>% return()",
  linters = pipe_return_linter()
)

# okay
code <- "function(x) {\n  y <- sum(x)\n  return(y)\n}"
writeLines(code)
lint(
  text = code,
  linters = pipe_return_linter()
)
```

---

pkg_testthat_linters *Testthat linters*

Description

Linters encouraging best practices within testthat suites.

Linters

The following linters are tagged with 'pkg_testthat':

- 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
- yoda_test_linter

See Also

- linters for a complete list of linters available in lintr.
- https://testthat.r-lib.org
- https://r-pkgs.org/testing-basics.html

`print_linter` *Block usage of print() for logging*

Description

The default print method for character vectors is appropriate for interactively inspecting objects, not for logging messages. Thus checked-in usage like `print(paste('Data has', nrow(DF), 'rows.'))` is better served by using `cat()`, e.g. `cat(sprintf('Data has %d rows.\n', nrow(DF)))` (noting that using `cat()` entails supplying your own line returns, and that `glue::glue()` might be preferable to `sprintf()` for constructing templated strings). Lastly, note that `message()` differs slightly from `cat()` in that it prints to `stderr` by default, not `stdout`, but is still a good option to consider for logging purposes.

Usage

```
print_linter()
```

Tags

best_practices, consistency

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "print('a')",
  linters = print_linter()
)

lint(
  text = "print(paste(x, 'y'))",
  linters = print_linter()
)

# okay
lint(
  text = "print(x)",
  linters = print_linter()
)
```

quotes_linter *Character string quote linter*

Description

Check that the desired quote delimiter is used for string constants.

Usage

```
quotes_linter(delimiter = c("\"", "'"))
```

Arguments

delimiter	Which quote delimiter to accept. Defaults to the tidyverse default of " (double-quoted strings).

Tags

configurable, consistency, default, readability, style

See Also

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

Examples

```
# will produce lints
lint(
  text = "c('a', 'b')",
  linters = quotes_linter()
)

# okay
lint(
  text = 'c("a", "b")',
  linters = quotes_linter()
)

code_lines <- "paste0(x, '\"this is fine\"')"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = quotes_linter()
)

# okay
lint(
  text = "c('a', 'b')",
```

```
  linters = quotes_linter(delimiter = "'")
)
```

`readability_linters` *Readability linters*

Description

Linters highlighting readability issues, such as missing whitespace.

Linters

The following linters are tagged with 'readability':

- `boolean_arithmetic_linter`
- `brace_linter`
- `coalesce_linter`
- `commas_linter`
- `commented_code_linter`
- `comparison_negation_linter`
- `conjunct_test_linter`
- `consecutive_assertion_linter`
- `consecutive_mutate_linter`
- `cyclocomp_linter`
- `empty_assignment_linter`
- `expect_length_linter`
- `expect_named_linter`
- `expect_not_linter`
- `expect_true_false_linter`
- `fixed_regex_linter`
- `for_loop_index_linter`
- `function_left_parentheses_linter`
- `function_return_linter`
- `if_not_else_linter`
- `if_switch_linter`
- `implicit_assignment_linter`
- `indentation_linter`
- `infix_spaces_linter`
- `inner_combine_linter`

- is_numeric_linter
- keyword_quote_linter
- length_levels_linter
- lengths_linter
- library_call_linter
- line_length_linter
- list2df_linter
- matrix_apply_linter
- nested_ifelse_linter
- nested_pipe_linter
- numeric_leading_zero_linter
- object_length_linter
- object_overwrite_linter
- object_usage_linter
- one_call_pipe_linter
- outer_negation_linter
- paren_body_linter
- pipe_call_linter
- pipe_consistency_linter
- pipe_continuation_linter
- quotes_linter
- redundant_equals_linter
- rep_len_linter
- repeat_linter
- sample_int_linter
- scalar_in_linter
- semicolon_linter
- sort_linter
- spaces_inside_linter
- spaces_left_parentheses_linter
- stopifnot_all_linter
- string_boundary_linter
- system_file_linter
- T_and_F_symbol_linter
- unnecessary_concatenation_linter
- unnecessary_lambda_linter
- unnecessary_nesting_linter
- unnecessary_placeholder_linter
- unreachable_code_linter
- which_grepl_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, call = parent.frame())
```

Arguments

`filename`	Source file to be linted.
`call`	Passed to malformed to ensure linear trace.

Details

The default linter_file name is `.lintr` but it can be changed with option `lintr.linter_file` or the environment variable `R_LINTR_LINTER_FILE` This file is a DCF file, see `base::read.dcf()` for details. Here is an example of a `.lintr` file:

```
linters: linters_with_defaults(
    any_duplicated_linter(),
    any_is_na_linter(),
    backport_linter("oldrel-4", except = c("R_user_dir", "str2lang")),
    line_length_linter(120L),
    missing_argument_linter(),
    unnecessary_concatenation_linter(allow_single_expression = FALSE),
    yoda_test_linter()
  )
exclusions: list(
    "inst/doc/creating_linters.R" = 1,
    "inst/example/bad.R",
    "tests/testthat/default_linter_testcode.R",
    "tests/testthat/dummy_packages"
  )
```

Experimentally, we also support keeping the config in a plain R file. By default we look for a file named `.lintr.R` (in the same directories where we search for `.lintr`). We are still deciding the future of config support in lintr, so user feedback is welcome. The advantage of R is that it maps more closely to how the configs are actually stored, whereas the DCF approach requires somewhat awkward formatting of parseable R code within valid DCF key-value pairs. The main disadvantage of the R file is it might be *too* flexible, with users tempted to write configs with side effects causing hard-to-detect bugs or like YAML could work, but require new dependencies and are harder to parse both programmatically and visually. Here is an example of a `.lintr.R` file:

```
linters <- linters_with_defaults(
    any_duplicated_linter(),
    any_is_na_linter(),
    backport_linter("oldrel-4", except = c("R_user_dir", "str2lang")),
    line_length_linter(120L),
    missing_argument_linter(),
    unnecessary_concatenation_linter(allow_single_expression = FALSE),
    yoda_test_linter()
  )
exclusions <- list(
    "inst/doc/creating_linters.R" = 1,
    "inst/example/bad.R",
    "tests/testthat/default_linter_testcode.R",
    "tests/testthat/dummy_packages"
  )
```

`redundant_equals_linter`

Block usage of `==`, `!=` *on logical vectors*

Description

Testing `x == TRUE` is redundant if `x` is a logical vector. Wherever this is used to improve readability, the solution should instead be to improve the naming of the object to better indicate that its contents are logical. This can be done using prefixes (is, has, can, etc.). For example, `is_child`, `has_parent_supervision`, `can_watch_horror_movie` clarify their logical nature, while `child`, `parent_supervision`, `watch_horror_movie` don't.

Usage

```
redundant_equals_linter()
```

Tags

best_practices, common_mistakes, efficiency, readability

See Also

- linters for a complete list of linters available in lintr.
- `outer_negation_linter()`

Examples

```
# will produce lints
lint(
  text = "if (any(x == TRUE)) 1",
  linters = redundant_equals_linter()
)

lint(
  text = "if (any(x != FALSE)) 0",
  linters = redundant_equals_linter()
)

lint(
  text = "dt[is_tall == FALSE, y]",
  linters = redundant_equals_linter()
)

# okay
lint(
  text = "if (any(x)) 1",
  linters = redundant_equals_linter()
)

lint(
  text = "if (!all(x)) 0",
  linters = redundant_equals_linter()
)

# in `{data.table}` semantics, `dt[x]` is a join, `dt[(x)]` is a subset
lint(
  text = "dt[(!is_tall), y]",
  linters = redundant_equals_linter()
)
```

---

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 be needed only 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, configurable, consistency, efficiency

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "ifelse(x >= 2.5, TRUE, FALSE)",
  linters = redundant_ifelse_linter()
)

lint(
  text = "ifelse(x < 2.5, 1L, 0L)",
  linters = redundant_ifelse_linter()
)

# okay
lint(
  text = "x >= 2.5",
  linters = redundant_ifelse_linter()
)

# Note that this is just to show the strict equivalent of the example above;
# converting to integer is often unnecessary and the logical vector itself
# should suffice.
lint(
  text = "as.integer(x < 2.5)",
  linters = redundant_ifelse_linter()
)

lint(
  text = "ifelse(x < 2.5, 1L, 0L)",
  linters = redundant_ifelse_linter(allow10 = TRUE)
)
```

regex_linters *Regular expression linters*

Description

Linters that examine the usage of regular expressions and functions executing them in user code.

Linters

The following linters are tagged with 'regex':

- fixed_regex_linter
- regex_subset_linter
- string_boundary_linter
- which_grepl_linter

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 grepv() returns the subset of the input that matches the pattern, e.g. grepv("[a-m]", letters) 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 grepv(pattern, x) 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, regex

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x[grep(pattern, x)]",
  linters = regex_subset_linter()
)

lint(
  text = "x[stringr::str_which(x, pattern)]",
  linters = regex_subset_linter()
)

# okay
lint(
  text = "grepv(pattern, x)",
  linters = regex_subset_linter()
)

lint(
  text = "stringr::str_subset(x, pattern)",
  linters = regex_subset_linter()
)
```

---

repeat_linter *Repeat linter*

Description

Check that `while (TRUE)` is not used for infinite loops. While this is valid R code, using `repeat {}` is more explicit.

Usage

```
repeat_linter()
```

Tags

readability, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "while (TRUE) { }",
  linters = repeat_linter()
)


# okay
lint(
  text = "repeat { }",
  linters = repeat_linter()
)
```

`rep_len_linter` *Require usage of rep_len(x, n) over rep(x, length.out = n)*

Description

`rep(x, length.out = n)` calls `rep_len(x, n)` "under the hood". The latter is thus more direct and equally readable.

Usage

```
rep_len_linter()
```

Tags

best_practices, consistency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "rep(1:3, length.out = 10)",
  linters = rep_len_linter()
)

# okay
lint(
```

```
  text = "rep_len(1:3, 10)",
  linters = rep_len_linter()
)

lint(
  text = "rep(1:3, each = 2L, length.out = 10L)",
  linters = rep_len_linter()
)
```

`return_linter` *Return linter*

Description

This linter checks functions' `return()` expressions.

Usage

```
return_linter(
  return_style = c("implicit", "explicit"),
  allow_implicit_else = TRUE,
  return_functions = NULL,
  except = NULL,
  except_regex = NULL
)
```

Arguments

`return_style`	Character string naming the return style. `"implicit"`, the default, enforces the Tidyverse guide recommendation to leave terminal returns implicit. `"explicit"` style requires that `return()` always be explicitly supplied.
`allow_implicit_else`	Logical, default `TRUE`. If `FALSE`, functions with a terminal `if` clause must always have an `else` clause, making the `NULL` alternative explicit if necessary. Similarly, functions with terminal `switch()` statements must have an explicit default case.
`return_functions`	Character vector of functions that are accepted as terminal calls when `return_style = "explicit"`. These are in addition to exit functions from base that are always allowed: `stop()`, `q()`, `quit()`, `invokeRestart()`, `tryInvokeRestart()`, `UseMethod()`, `NextMethod()`, `standardGeneric()`, `callNextMethod()`, `.C()`, `.Call()`, `.External()`, and `.Fortran()`.
`except`, `except_regex`	Character vector of functions that are not checked when `return_style = "explicit"`. These are in addition to namespace hook functions that are never checked: `.onLoad()`, `.onUnload()`, `.onAttach()`, `.onDetach()`, `.Last.lib()`, `.First()` and `.Last()`. `except` matches function names exactly, while `except_regex` does exclusion by pattern matching with `rex::re_matches()`.

Tags

configurable, default, style

See Also

- linters for a complete list of linters available in lintr.
- https://style.tidyverse.org/functions.html?q=return#return

Examples

```
# will produce lints
code <- "function(x) {\n  return(x + 1)\n}"
writeLines(code)
lint(
  text = code,
  linters = return_linter()
)

code <- "function(x) {\n  x + 1\n}"
writeLines(code)
lint(
  text = code,
  linters = return_linter(return_style = "explicit")
)

code <- "function(x) {\n  if (x > 0) 2\n}"
writeLines(code)
lint(
  text = code,
  linters = return_linter(allow_implicit_else = FALSE)
)

# okay
code <- "function(x) {\n  x + 1\n}"
writeLines(code)
lint(
  text = code,
  linters = return_linter()
)

code <- "function(x) {\n  return(x + 1)\n}"
writeLines(code)
lint(
  text = code,
  linters = return_linter(return_style = "explicit")
)

code <- "function(x) {\n  if (x > 0) 2 else NULL\n}"
writeLines(code)
lint(
  text = code,
  linters = return_linter(allow_implicit_else = FALSE)
```

```
)
```

`robustness_linters` *Robustness linters*

Description

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

Linters

The following linters are tagged with 'robustness':

- absolute_path_linter
- all_equal_linter
- backport_linter
- class_equals_linter
- download_file_linter
- equals_na_linter
- for_loop_index_linter
- missing_package_linter
- namespace_linter
- nonportable_path_linter
- object_overwrite_linter
- routine_registration_linter
- sample_int_linter
- seq_linter
- strings_as_factors_linter
- T_and_F_symbol_linter
- terminal_close_linter
- undesirable_function_linter
- undesirable_operator_linter

See Also

linters for a complete list of linters available in lintr.

`routine_registration_linter` *Identify unregistered native routines*

Description

It is preferable to register routines for efficiency and safety.

Usage

```
routine_registration_linter()
```

Tags

best_practices, efficiency, robustness

See Also

- linters for a complete list of linters available in lintr.
- https://cran.r-project.org/doc/manuals/r-release/R-exts.html#Registering-native-routines

Examples

```
# will produce lints
lint(
  text = '.Call("cpp_routine", PACKAGE = "mypkg")',
  linters = routine_registration_linter()
)

lint(
  text = '.Fortran("f_routine", PACKAGE = "mypkg")',
  linters = routine_registration_linter()
)

# okay
lint(
  text = ".Call(cpp_routine)",
  linters = routine_registration_linter()
)

lint(
  text = ".Fortran(f_routine)",
  linters = routine_registration_linter()
)
```

`sample_int_linter` *Require usage of sample.int(n, m, ...) over sample(1:n, m, ...)*

Description

`sample.int()` is preferable to `sample()` for the case of sampling numbers between 1 and `n`. `sample` calls `sample.int()` "under the hood".

Usage

```
sample_int_linter()
```

Tags

efficiency, readability, robustness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "sample(1:10, 2)",
  linters = sample_int_linter()
)

lint(
  text = "sample(seq(4), 2)",
  linters = sample_int_linter()
)

lint(
  text = "sample(seq_len(8), 2)",
  linters = sample_int_linter()
)

# okay
lint(
  text = "sample(seq(1, 5, by = 2), 2)",
  linters = sample_int_linter()
)

lint(
  text = "sample(letters, 2)",
  linters = sample_int_linter()
)
```

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

scalar_in_linter *Block usage like x %in% "a"*

Description

`vector %in% set` is appropriate for matching a vector to a set, but if that set has size 1, == is more appropriate. However, if `vector` has also size 1 and can be NA, the use of == should be accompanied by extra protection for the missing case (for example, `isTRUE(NA == "arg")` or `!is.na(x) && x == "arg"`).

Usage

```
scalar_in_linter(in_operators = NULL)
```

Arguments

in_operators	Character vector of additional infix operators that behave like the %in% operator, e.g. {data.table}'s %chin% operator.

Details

`scalar %in% vector` is OK, because the alternative (`any(vector == scalar)`) is more circuitous & potentially less clear.

Tags

best_practices, configurable, consistency, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x %in% 1L",
  linters = scalar_in_linter()
)

lint(
  text = "x %chin% 'a'",
  linters = scalar_in_linter(in_operators = "%chin%")
)

# okay
lint(
  text = "x %in% 1:10",
  linters = scalar_in_linter()
)
```

`semicolon_linter` *Semicolon linter*

Description

Check that no semicolons terminate expressions.

Usage

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

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.

Tags

configurable, default, readability, style

See Also

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

Examples

```
# will produce lints
lint(
  text = "a <- 1;",
  linters = semicolon_linter()
)

lint(
  text = "a <- 1; b <- 1",
  linters = semicolon_linter()
)

lint(
  text = "function() { a <- 1; b <- 1 }",
  linters = semicolon_linter()
)

# okay
lint(
  text = "a <- 1",
  linters = semicolon_linter()
)

lint(
  text = "a <- 1;",
  linters = semicolon_linter(allow_trailing = TRUE)
)

code_lines <- "a <- 1\nb <- 1"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = semicolon_linter()
)

lint(
  text = "a <- 1; b <- 1",
  linters = semicolon_linter(allow_compound = TRUE)
)

code_lines <- "function() { \n  a <- 1\n  b <- 1\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = semicolon_linter()
)
```

`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. Instead, it is safer to use `base::seq_len()` (to create a sequence of a specified *length*) or `base::seq_along()` (to create a sequence *along* an object).

Tags

best_practices, consistency, default, efficiency, robustness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "seq(length(x))",
  linters = seq_linter()
)

lint(
  text = "1:nrow(x)",
  linters = seq_linter()
)

lint(
  text = "dplyr::mutate(x, .id = 1:n())",
  linters = seq_linter()
)

lint(
  text = "seq_len(length(x))",
  linters = seq_linter()
)

lint(
  text = "unlist(lapply(x, seq_len))",
  linters = seq_linter()
)
```

```
# okay
lint(
  text = "seq_along(x)",
  linters = seq_linter()
)

lint(
  text = "seq_len(nrow(x))",
  linters = seq_linter()
)

lint(
  text = "dplyr::mutate(x, .id = seq_len(n()))",
  linters = seq_linter()
)

lint(
  text = "seq_along(x)",
  linters = seq_linter()
)

lint(
  text = "sequence(x)",
  linters = seq_linter()
)
```

`sort_linter` *Check for common mistakes around sorting vectors*

Description

This linter checks for some common mistakes when using `order()` or `sort()`.

Usage

```
sort_linter()
```

Details

First, it requires usage of `sort()` over `.[order(.)]`.

`sort()` is the dedicated option to sort a list or vector. It is more legible and around twice as fast as `.[order(.)]`, with the gap in performance growing with the vector size.

Second, it requires usage of `is.unsorted()` over equivalents using `sort()`.

The base function `is.unsorted()` exists to test the sortedness of a vector. Prefer it to inefficient and less-readable equivalents like `x != sort(x)`. The same goes for checking `x == sort(x)` – use `!is.unsorted(x)` instead.

Moreover, use of `x == sort(x)` can be risky because `sort()` drops missing elements by default, meaning `==` might end up trying to compare vectors of differing lengths.

Tags

best_practices, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x[order(x)]",
  linters = sort_linter()
)

lint(
  text = "x[order(x, decreasing = TRUE)]",
  linters = sort_linter()
)

lint(
  text = "sort(x) == x",
  linters = sort_linter()
)

# okay
lint(
  text = "x[sample(order(x))]",
  linters = sort_linter()
)

lint(
  text = "y[order(x)]",
  linters = sort_linter()
)

lint(
  text = "sort(x, decreasing = TRUE) == x",
  linters = sort_linter()
)

# If you are sorting several objects based on the order of one of them, such
# as:
x <- sample(1:26)
y <- letters
newx <- x[order(x)]
newy <- y[order(x)]
# This will be flagged by the linter. However, in this very specific case,
# it would be clearer and more efficient to run order() once and assign it
# to an object, rather than mix and match order() and sort()
index <- order(x)
newx <- x[index]
```

```
newy <- y[index]
```

`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

Examples

```
# will produce lints
lint(
  text = "c( TRUE, FALSE )",
  linters = spaces_inside_linter()
)

lint(
  text = "x[ 1L ]",
  linters = spaces_inside_linter()
)

# okay
lint(
  text = "c(TRUE, FALSE)",
  linters = spaces_inside_linter()
)

lint(
  text = "x[1L]",
  linters = spaces_inside_linter()
)
```

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
- function_left_parentheses_linter()

Examples

```
# will produce lints
lint(
  text = "if(TRUE) x else y",
  linters = spaces_left_parentheses_linter()
)

# okay
lint(
  text = "if (TRUE) x else y",
  linters = spaces_left_parentheses_linter()
)
```

sprintf_linter *Require correct* sprintf() *calls*

Description

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

Usage

```
sprintf_linter()
```

Details

gettextf() calls are also included, since gettextf() is a thin wrapper around sprintf().

Tags

common_mistakes, correctness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = 'sprintf("hello %s %s %d", x, y)',
  linters = sprintf_linter()
)

lint(
  text = 'sprintf("hello")',
  linters = sprintf_linter()
)

# okay
lint(
  text = 'sprintf("hello %s %s %d", x, y, z)',
  linters = sprintf_linter()
)

lint(
  text = 'sprintf("hello %s %s %d", x, y, ...)',
  linters = sprintf_linter()
)
```

---

stopifnot_all_linter *Block usage of all() within stopifnot()*

Description

stopifnot(A) actually checks all(A) "under the hood" if A is a vector, and produces a better error message than stopifnot(all(A)) does.

Usage

```
stopifnot_all_linter()
```

Tags

best_practices, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "stopifnot(all(x > 0))",
  linters = stopifnot_all_linter()
)

lint(
  text = "stopifnot(y > 3, all(x < 0))",
  linters = stopifnot_all_linter()
)

# okay
lint(
  text = "stopifnot(is.null(x) || all(x > 0))",
  linters = stopifnot_all_linter()
)

lint(
  text = "assert_that(all(x > 0))",
  linters = stopifnot_all_linter()
)
```

---

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.

Examples

```
# will produce lints
lint(
  text = 'data.frame(x = "a")',
  linters = strings_as_factors_linter()
)

# okay
lint(
  text = 'data.frame(x = "a", stringsAsFactors = TRUE)',
  linters = strings_as_factors_linter()
)

lint(
  text = 'data.frame(x = "a", stringsAsFactors = FALSE)',
  linters = strings_as_factors_linter()
)

lint(
  text = "data.frame(x = 1.2)",
  linters = strings_as_factors_linter()
)
```

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

Description

base::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 conciseness offered by grepl() whereby NA input maps to FALSE output, which doesn't have a direct equivalent with startsWith() or endsWith().

Details

Ditto for using base::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 users.

Tags

configurable, efficiency, readability, regex

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = 'grepl("^a", x)',
  linters = string_boundary_linter()
)

lint(
  text = 'grepl("z$", x)',
  linters = string_boundary_linter()
)

# okay
lint(
```

```
  text = 'startsWith(x, "a")',
  linters = string_boundary_linter()
)

lint(
  text = 'endsWith(x, "z")',
  linters = string_boundary_linter()
)

# If missing values are present, the suggested alternative wouldn't be strictly
# equivalent, so this linter can also be turned off in such cases.
lint(
  text = 'grepl("z$", x)',
  linters = string_boundary_linter(allow_grepl = TRUE)
)
```

`style_linters` *Style linters*

Description

Linters highlighting code style issues.

Linters

The following linters are tagged with 'style':

- assignment_linter
- brace_linter
- commas_linter
- commented_code_linter
- condition_call_linter
- consecutive_assertion_linter
- cyclocomp_linter
- function_argument_linter
- function_left_parentheses_linter
- implicit_assignment_linter
- implicit_integer_linter
- indentation_linter
- infix_spaces_linter
- keyword_quote_linter
- library_call_linter
- line_length_linter

- numeric_leading_zero_linter
- object_length_linter
- object_name_linter
- object_usage_linter
- one_call_pipe_linter
- package_hooks_linter
- paren_body_linter
- pipe_call_linter
- pipe_consistency_linter
- pipe_continuation_linter
- quotes_linter
- repeat_linter
- return_linter
- semicolon_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
- unnecessary_concatenation_linter
- whitespace_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.

Examples

```
# will produce lints
lint(
  text = 'system.file(file.path("path", "to", "data"), package = "foo")',
  linters = system_file_linter()
)

lint(
  text = 'file.path(system.file(package = "foo"), "path", "to", "data")',
  linters = system_file_linter()
)

# okay
lint(
  text = 'system.file("path", "to", "data", package = "foo")',
  linters = system_file_linter()
)
```

---

terminal_close_linter *Prohibit close() from terminating a function definition*

Description

Functions that end in `close(x)` are almost always better written by using `on.exit(close(x))` close to where `x` is defined and/or opened.

Usage

```
terminal_close_linter()
```

Tags

best_practices, robustness

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
code <- paste(
  "f <- function(fl) {",
  "  conn <- file(fl, open = 'r')",
  "  readLines(conn)",
  "  close(conn)",
  "}",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = terminal_close_linter()
)

# okay
code <- paste(
  "f <- function(fl) {",
  "  conn <- file(fl, open = 'r')",
  "  on.exit(close(conn))",
  "  readLines(conn)",
  "}",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = terminal_close_linter()
)
```

---

tidy_design_linters *Tidyverse design linters*

Description

Linters based on guidelines described in the 'Tidy design principles' book.

Linters

The following linters are tagged with 'tidy_design':

- condition_call_linter

See Also

- linters for a complete list of linters available in lintr.
- https://design.tidyverse.org/

`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"), except_regex = NULL)
```

Arguments

`todo`	Vector of case-insensitive strings that identify TODO comments.
`except_regex`	Vector of case-sensitive regular expressions that identify *valid* TODO comments.

Tags

configurable, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x + y # TOODOO",
  linters = todo_comment_linter(todo = "toodoo")
)

lint(
  text = "pi <- 1.0 # FIIXMEE",
  linters = todo_comment_linter(todo = "fiixmee")
)

lint(
  text = "x <- TRUE # TOODOO(#1234): Fix this hack.",
  linters = todo_comment_linter()
)

# okay
lint(
  text = "x + y # my informative comment",
  linters = todo_comment_linter()
)
```

```
lint(
  text = "pi <- 3.14",
  linters = todo_comment_linter()
)

lint(
  text = "x <- TRUE",
  linters = todo_comment_linter()
)

lint(
  text = "x <- TRUE # TODO(#1234): Fix this hack.",
  linters = todo_comment_linter(except_regex = "TODO\\(#[0-9]+\\):")
)
```

`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.

Examples

```
# will produce lints
f <- tempfile()
cat("x <- 1\n\n", file = f)
writeLines(readChar(f, file.size(f)))
lint(
  filename = f,
  linters = trailing_blank_lines_linter()
)
unlink(f)

# okay
cat("x <- 1\n", file = f)
```

```
writeLines(readChar(f, file.size(f)))
lint(
  filename = f,
  linters = trailing_blank_lines_linter()
)
unlink(f)
```

`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

configurable, default, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x <- 1.2  ",
  linters = trailing_whitespace_linter()
)

code_lines <- "a <- TRUE\n \nb <- FALSE"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = trailing_whitespace_linter()
)
```

```
# okay
lint(
  text = "x <- 1.2",
  linters = trailing_whitespace_linter()
)

lint(
  text = "x <- 1.2  # comment about this assignment",
  linters = trailing_whitespace_linter()
)

code_lines <- "a <- TRUE\n \nb <- FALSE"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = trailing_whitespace_linter(allow_empty_lines = TRUE)
)
```

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

Description

Although they can be synonyms, avoid the symbols `T` and `F`, and use `TRUE` and `FALSE`, respectively, instead. `T` and `F` are not reserved keywords and can be assigned to any other values.

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

Examples

```
# will produce lints
lint(
  text = "x <- T; y <- F",
  linters = T_and_F_symbol_linter()
)
```

```
lint(
  text = "T = 1.2; F = 2.4",
  linters = T_and_F_symbol_linter()
)

# okay
lint(
  text = "x <- c(TRUE, FALSE)",
  linters = T_and_F_symbol_linter()
)

lint(
  text = "t = 1.2; f = 2.4",
  linters = T_and_F_symbol_linter()
)
```

`undesirable_function_linter` *Undesirable function linter*

Description

Report the use of undesirable functions and suggest an alternative.

Usage

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

Arguments

`fun` Character vector of undesirable function names. Input can be any of three types:

- Unnamed entries must be a character string specifying an undesirable function.
- For named entries, the name specifies the undesirable function.
  - If the entry is a character string, it is used as a description of why a given function is undesirable
  - Otherwise, entries should be missing (`NA`) A generic message that the named function is undesirable is used if no specific description is provided. Input can also be a list of character strings for convenience.

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, robustness, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# defaults for which functions are considered undesirable
names(default_undesirable_functions)

# will produce lints
lint(
  text = "sapply(x, mean)",
  linters = undesirable_function_linter()
)

lint(
  text = "log10(x)",
  linters = undesirable_function_linter(fun = c("log10" = NA))
)

lint(
  text = "log10(x)",
  linters = undesirable_function_linter(fun = c("log10" = "use log()"))
)

lint(
  text = 'dir <- "path/to/a/directory"',
  linters = undesirable_function_linter(fun = c("dir" = NA))
)


lint(
  text = 'dir <- "path/to/a/directory"',
  linters = undesirable_function_linter(fun = "dir")
)

# okay
lint(
  text = "vapply(x, mean, FUN.VALUE = numeric(1))",
  linters = undesirable_function_linter()
)

lint(
  text = "log(x, base = 10)",
  linters = undesirable_function_linter(fun = c("log10" = "use log()"))
)

lint(
  text = 'dir <- "path/to/a/directory"',
```

```
  linters = undesirable_function_linter(fun = c("dir" = NA), symbol_is_undesirable = FALSE)
)

lint(
  text = 'dir <- "path/to/a/directory"',
  linters = undesirable_function_linter(fun = "dir", symbol_is_undesirable = FALSE)
)
```

`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,
  call_is_undesirable = TRUE
)
```

Arguments

op	Character vector of undesirable operators. Input can be any of three types: • Unnamed entries must be a character string specifying an undesirable operator. • For named entries, the name specifies the undesirable operator. – If the entry is a character string, it is used as a description of why a given operator is undesirable – Otherwise, entries should be missing (`NA`) A generic message that the named operator is undesirable is used if no specific description is provided. Input can also be a list of character strings for convenience. Defaults to default_undesirable_operators. To make small customizations to this list, use `modify_defaults()`.
call_is_undesirable	Logical, default `TRUE`. Should lints also be produced for prefix-style usage of the operators provided in `op`?

Tags

best_practices, configurable, robustness, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# defaults for which functions are considered undesirable
names(default_undesirable_operators)

# will produce lints
lint(
  text = "a <<- log(10)",
  linters = undesirable_operator_linter()
)

lint(
  text = "mtcars$wt",
 linters = undesirable_operator_linter(op = c("$" = "As an alternative, use the `[[` accessor."))
)

lint(
  text = "`:::`(utils, hasName)",
  linters = undesirable_operator_linter()
)

lint(
  text = "mtcars$wt",
  linters = undesirable_operator_linter("$")
)

# okay
lint(
  text = "a <- log(10)",
  linters = undesirable_operator_linter()
)
lint(
  text = 'mtcars[["wt"]]',
  linters = undesirable_operator_linter(op = c("$" = NA))
)

lint(
  text = 'mtcars[["wt"]]',
 linters = undesirable_operator_linter(op = c("$" = "As an alternative, use the `[[` accessor."))
)

lint(
  text = "`:::`(utils, hasName)",
  linters = undesirable_operator_linter(call_is_undesirable = FALSE)
)

lint(
  text = 'mtcars[["wt"]]',
  linters = undesirable_operator_linter("$")
)
```

unnecessary_concatenation_linter
Unneeded concatenation linter

Description

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

Usage

```
unnecessary_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 the more readable `as.vector()` instead. `as.vector()` is not always preferable, for example with environments (especially, `R6` objects), in which case `list()` is the better alternative.

Tags

configurable, efficiency, readability, style

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x <- c()",
  linters = unnecessary_concatenation_linter()
)

lint(
  text = "x <- c(TRUE)",
  linters = unnecessary_concatenation_linter()
)

lint(
  text = "x <- c(1.5 + 2.5)",
  linters = unnecessary_concatenation_linter(allow_single_expression = FALSE)
)

# okay
```

```
lint(
  text = "x <- NULL",
  linters = unnecessary_concatenation_linter()
)

# In case the intent here was to seed a vector of known size
lint(
  text = "x <- integer(4L)",
  linters = unnecessary_concatenation_linter()
)

lint(
  text = "x <- TRUE",
  linters = unnecessary_concatenation_linter()
)

lint(
  text = "x <- c(1.5 + 2.5)",
  linters = unnecessary_concatenation_linter(allow_single_expression = TRUE)
)
```

`unnecessary_lambda_linter` *Block usage of anonymous functions in iteration functions when unnecessary*

Description

Using an anonymous function in, e.g., `lapply()` is not always necessary, e.g. `lapply(DF, sum)` is the same as `lapply(DF, function(x) sum(x))` and the former is more readable.

Usage

```
unnecessary_lambda_linter(allow_comparison = FALSE)
```

Arguments

`allow_comparison`
: Logical, default `FALSE`. If `TRUE`, lambdas like `function(x) foo(x) == 2`, where `foo` can be extracted to the "mapping" function and `==` vectorized instead of called repeatedly, are linted.

Details

Cases like `lapply(x, \(xi) grep("ptn", xi))` are excluded because, though the anonymous function *can* be avoided, doing so is not always more readable.

Tags

best_practices, configurable, efficiency, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "lapply(list(1:3, 2:4), function(xi) sum(xi))",
  linters = unnecessary_lambda_linter()
)

lint(
  text = "sapply(x, function(xi) xi == 2)",
  linters = unnecessary_lambda_linter()
)

lint(
  text = "sapply(x, function(xi) sum(xi) > 0)",
  linters = unnecessary_lambda_linter()
)

# okay
lint(
  text = "lapply(list(1:3, 2:4), sum)",
  linters = unnecessary_lambda_linter()
)

lint(
  text = 'lapply(x, function(xi) grep("ptn", xi))',
  linters = unnecessary_lambda_linter()
)

lint(
  text = "lapply(x, function(xi) data.frame(col = xi))",
  linters = unnecessary_lambda_linter()
)

lint(
  text = "sapply(x, function(xi) xi == 2)",
  linters = unnecessary_lambda_linter(allow_comparison = TRUE)
)

lint(
  text = "sapply(x, function(xi) sum(xi) > 0)",
  linters = unnecessary_lambda_linter(allow_comparison = TRUE)
)

lint(
```

```
  text = "sapply(x, function(xi) sum(abs(xi)) > 10)",
  linters = unnecessary_lambda_linter()
)

lint(
  text = "sapply(x, sum) > 0",
  linters = unnecessary_lambda_linter()
)
```

`unnecessary_nesting_linter` *Block instances of unnecessary nesting*

Description

Excessive nesting harms readability. Use helper functions or early returns to reduce nesting wherever possible.

Usage

```
unnecessary_nesting_linter(
  allow_assignment = TRUE,
 allow_functions = c("switch", "try", "tryCatch", "withCallingHandlers", "quote",
  "expression", "bquote", "substitute", "with_parameters_test_that", "reactive",
  "observe", "observeEvent", "renderCachedPlot", "renderDataTable", "renderImage",
   "renderPlot", "renderPrint", "renderTable", "renderText", "renderUI"),
  branch_exit_calls = character()
)
```

Arguments

`allow_assignment`

Logical, default `TRUE`, in which case braced expressions consisting only of a single assignment are skipped. if `FALSE`, all braced expressions with only one child expression are linted. The `TRUE` case facilitates interaction with `implicit_assignment_linter()` for certain cases where an implicit assignment is necessary, so a braced assignment is used to further distinguish the assignment. See examples.

`allow_functions`

Character vector of functions which always allow one-child braced expressions. `testthat::test_that()` is always allowed because testthat requires a braced expression in its code argument. The other defaults similarly compute on expressions in a way which is worth highlighting by em-bracing them, even if there is only one expression, while `switch()` is allowed for its use as a control flow analogous to `if`/`else`.]

branch_exit_calls
: Character vector of functions which are considered as "exiting" a branch for the purpose of recommending removing nesting in a branch *lacking* an exit call when the other branch terminates with one. Calls which always interrupt or quit the current call or R session, e.g. `stop()` and `q()`, are always included.

Tags

best_practices, configurable, consistency, readability

See Also

- `cyclocomp_linter()` for another linter that penalizes overly complex code.
- linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
code <- "if (A) {\n  stop('A is bad!')\n} else {\n  do_good()\n}"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

code <- "tryCatch(\n  {\n    foo()\n  },\n  error = identity\n)"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

code <- "expect_warning(\n  {\n    x <- foo()\n  },\n  'warned'\n)"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter(allow_assignment = FALSE)
)

code <- "if (x) { \n  if (y) { \n   return(1L) \n  } \n}"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

lint(
  text = "my_quote({x})",
  linters = unnecessary_nesting_linter()
)

code <- paste(
```

```
  "if (A) {",
  "  stop('A is bad because a.')",
  "} else {",
  "  warning('!A requires caution.')",
  "}",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

# okay
code <- "if (A) {\n  stop('A is bad because a.')\n} else {\n  stop('!A is bad too.')\n}"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

code <- "capture.output({\n  foo()\n})"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

code <- "expect_warning(\n  {\n    x <- foo()\n  },\n  'warned'\n)"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

code <- "if (x && y) { \n  return(1L) \n}"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

code <- "if (x) { \n  y <- x + 1L\n  if (y) { \n   return(1L) \n  } \n}"
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter()
)

lint(
  text = "my_quote({x})",
  linters = unnecessary_nesting_linter(allow_functions = "my_quote")
)
```

```
code <- paste(
  "if (A) {",
  "  stop('A is bad because a.')",
  "} else {",
  "  warning('!A requires caution.')",
  "}",
  sep = "\n"
)
writeLines(code)
lint(
  text = code,
  linters = unnecessary_nesting_linter(branch_exit_calls = c("stop", "warning"))
)
```

`unnecessary_placeholder_linter` *Block usage of pipeline placeholders if unnecessary*

Description

The argument placeholder `.` in magrittr pipelines is unnecessary if passed as the first positional argument; using it can cause confusion and impacts readability.

Usage

```
unnecessary_placeholder_linter()
```

Details

This is true for forward (`%>%`), assignment (`%<>%`), and tee (`%T>%`) operators.

Tags

best_practices, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "x %>% sum(., na.rm = TRUE)",
  linters = unnecessary_placeholder_linter()
)
```

```
# okay
lint(
  text = "x %>% sum(na.rm = TRUE)",
  linters = unnecessary_placeholder_linter()
)

lint(
  text = "x %>% lm(data = ., y ~ z)",
  linters = unnecessary_placeholder_linter()
)

lint(
  text = "x %>% outer(., .)",
  linters = unnecessary_placeholder_linter()
)
```

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

Description

Code after e.g. a `return()` or `stop()` or in deterministically false conditional loops like `if (FALSE)` 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(
  allow_comment_regex = getOption("covr.exclude_end", "# nocov end")
)
```

Arguments

`allow_comment_regex`
: Character vector of regular expressions which identify comments to exclude when finding unreachable terminal comments. By default, this includes the default "skip region" end marker for `{covr}` (option "covr.exclude_end", or `"# nocov end"` if unset). The end marker for `{lintr}` (`settings$exclude_end`) is always included. Note that the regexes should include the initial comment character #.

Tags

best_practices, configurable, readability

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
code_lines <- "f <- function() {\n  return(1 + 1)\n  2 + 2\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "if (FALSE) {\n 2 + 2\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "while (FALSE) {\n 2 + 2\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "f <- function() {\n  return(1)\n  # end skip\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

# okay
code_lines <- "f <- function() {\n  return(1 + 1)\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "if (foo) {\n 2 + 2\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "while (foo) {\n 2 + 2\n}"
writeLines(code_lines)
lint(
```

```
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "f <- function() {\n  return(1)\n  # end skip\n}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter(allow_comment_regex = "# end skip")
)
```

`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"),
  interpret_glue = TRUE
)
```

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 attaching it.
`except_packages`	Character vector of packages that are ignored. These are usually attached for their side effects.
`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

best_practices, common_mistakes, configurable, executing

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
code_lines <- "library(dplyr)\n1 + 1"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unused_import_linter()
)

code_lines <- "library(dplyr)\ndplyr::tibble(a = 1)"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unused_import_linter()
)

# okay
code_lines <- "library(dplyr)\ntibble(a = 1)"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unused_import_linter()
)

code_lines <- "library(dplyr)\ndplyr::tibble(a = 1)"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unused_import_linter(allow_ns_usage = TRUE)
)
```

---

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 `all_linters()`.

Value

Path to the generated configuration, invisibly.

See Also

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

Examples

```
if (FALSE) {
  # use the default set of linters
  lintr::use_lintr()
  # or try all linters
  lintr::use_lintr(type = "full")

  # then
  lintr::lint_dir()
}
```

---

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 of 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, common_mistakes, default, efficiency

See Also

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

Examples

```
# will produce lints
lint(
  text = "if (TRUE & FALSE) 1",
  linters = vector_logic_linter()
)

lint(
  text = "if (TRUE && (TRUE | FALSE)) 4",
  linters = vector_logic_linter()
)

lint(
  text = "filter(x, A && B)",
  linters = vector_logic_linter()
)

# okay
lint(
  text = "if (TRUE && FALSE) 1",
  linters = vector_logic_linter()
)

lint(
  text = "if (TRUE && (TRUE || FALSE)) 4",
  linters = vector_logic_linter()
)

lint(
  text = "filter(x, A & B)",
  linters = vector_logic_linter()
)
```

---

which_grepl_linter *Require usage of grep over which(grepl(.))*

Description

`which(grepl(pattern, x))` is the same as `grep(pattern, x)`, but harder to read and requires two passes over the vector.

Usage

```
which_grepl_linter()
```

Tags

consistency, efficiency, readability, regex

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "which(grepl('^a', x))",
  linters = which_grepl_linter()
)

# okay
lint(
  text = "which(grepl('^a', x) | grepl('^b', x))",
  linters = which_grepl_linter()
)
```

---

whitespace_linter *Whitespace linter*

Description

Check that the correct character is used for indentation.

Usage

```
whitespace_linter()
```

Details

Currently, only supports linting in the presence of tabs.

Much ink has been spilled on this topic, and we encourage you to check out references for more information.

Tags

consistency, default, style

References

- https://www.jwz.org/doc/tabs-vs-spaces.html
- https://blog.codinghorror.com/death-to-the-space-infidels/

See Also

linters for a complete list of linters available in lintr.

Examples

```
# will produce lints
lint(
  text = "\tx",
  linters = whitespace_linter()
)

# okay
lint(
  text = "  x",
  linters = whitespace_linter()
)
```

---

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).

---

xp_call_name *Get the name of the function matched by an XPath*

Description

Often, it is more helpful to tailor the message of a lint to record which function was matched by the lint logic. This function encapsulates the logic to pull out the matched call in common situations.

Usage

```
xp_call_name(expr, depth = 1L, condition = NULL)
```

Arguments

expr	An xml_node or xml_nodeset, e.g. from xml2::xml_find_all().
depth	Integer, default 1L. How deep in the AST represented by expr should we look to find the call? By default, we assume expr is matched to an <expr> node under which the corresponding <SYMBOL_FUNCTION_CALL> node is found directly. depth = 0L means expr is matched directly to the SYMBOL_FUNCTION_CALL; depth > 1L means depth total <expr> nodes must be traversed before finding the call.
condition	An additional (XPath condition on the SYMBOL_FUNCTION_CALL required for a match. The default (NULL) is no condition. See examples.

Examples

```
xml_from_code <- function(str) {
  xml2::read_xml(xmlparsedata::xml_parse_data(parse(text = str, keep.source = TRUE)))
}
xml <- xml_from_code("sum(1:10)")
xp_call_name(xml, depth = 2L)

xp_call_name(xml2::xml_find_first(xml, "expr"))

xml <- xml_from_code(c("sum(1:10)", "sd(1:10)"))
xp_call_name(xml, depth = 2L, condition = "text() = 'sum'")
```

---

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, pkg_testthat, readability

See Also

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

Examples

```
# will produce lints
lint(
  text = "expect_equal(2, x)",
  linters = yoda_test_linter()
)

lint(
  text = 'expect_identical("a", x)',
  linters = yoda_test_linter()
)

# okay
lint(
  text = "expect_equal(x, 2)",
  linters = yoda_test_linter()
)

lint(
  text = 'expect_identical(x, "a")',
  linters = yoda_test_linter()
)
```

Index