Package ‘lintr’

November 7, 2023

Title  A 'Linter' for R Code
Version  3.1.1
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
BugReports  https://github.com/r-lib/lintr/issues
Depends  R (>= 3.5)
Imports  backports (>= 1.1.7),
         codetools,
         cyclocomp,
         digest,
         glue,
         knitr,
         rex,
         stats,
         utils,
         xml2 (>= 1.0.0),
         xmlparsedata (>= 1.0.5)
Suggests  bookdown,
          crayon,
          httr (>= 1.2.1),
          jsonlite,
          mockery,
          patrick,
          rlang,
          rstudioapi (>= 0.2),
          testthat (>= 3.1.5),
          tibble,
          tufte,
          withr (>= 2.5.0)
Enhances  data.table
VignetteBuilder  knitr
Config/Needs/website tidyverse/tidytemplate
Config/testthat/edition 3
Encoding UTF-8
Roxygen list(markdown = TRUE)
RoxygenNote 7.2.3
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'
  '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_equal_linter.R'
  'commas_linter.R'
  'comment_linters.R'
  'comments. R'
  'condition_message_linter.R'
  'conjunct_test_linter.R'
  'consecutive_assertion_linter.R'
  'cyclocomp_linter.R'
  'declared_functions.R'
  'deprecated.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'
  'extraction_operator_linter.R'
  'fixed_regex_linter.R'
  'for_loop_index_linter.R'
  'function_argument_linter.R'
  'function_left_parentheses_linter.R'
R topics documented:

- 'semicolon_linter.R'
- 'seq_linter.R'
- 'settings.R'
- 'settings_utils.R'
- 'shared_constants.R'
- 'sort_linter.R'
- 'spaces_inside_linter.R'
- 'spaces_left_parentheses_linter.R'
- 'sprintf_linter.R'
- 'string_boundary_linter.R'
- 'strings_as_factors_linter.R'
- 'system_file_linter.R'
- 'trailing_blank_lines_linter.R'
- 'trailing_whitespace_linter.R'
- 'tree_utils.R'
- 'undesirable_function_linter.R'
- 'undesirable_operator_linter.R'
- 'unnecessary_concatenation_linter.R'
- 'unnecessary_lambda_linter.R'
- 'unnecessary_nested_if_linter.R'
- 'unnecessary_placeholder_linter.R'
- 'unreachable_code_linter.R'
- 'unused_import_linter.R'
- 'use_lintr.R'
- 'vector_logic_linter.R'
- 'whitespace_linter.R'
- 'with.R'
- 'with_id.R'
- 'xml_nodes_to_lints.R'
- 'yoda_test_linter.R'
- 'zzz.R'

Language en-US

R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute_path_linter</td>
<td>7</td>
</tr>
<tr>
<td>all_linters</td>
<td>8</td>
</tr>
<tr>
<td>all_undesirable_functions</td>
<td>9</td>
</tr>
<tr>
<td>any_duplicated_linter</td>
<td>10</td>
</tr>
<tr>
<td>any_is_na_linter</td>
<td>11</td>
</tr>
<tr>
<td>assignment_linter</td>
<td>12</td>
</tr>
<tr>
<td>available_linters</td>
<td>14</td>
</tr>
<tr>
<td>backport_linter</td>
<td>15</td>
</tr>
<tr>
<td>best_practices_linters</td>
<td>16</td>
</tr>
<tr>
<td>boolean_arithmetic_linter</td>
<td>18</td>
</tr>
<tr>
<td>brace_linter</td>
<td>19</td>
</tr>
<tr>
<td>checkstyle_output</td>
<td>20</td>
</tr>
<tr>
<td>class_equals_linter</td>
<td>21</td>
</tr>
<tr>
<td>clear_cache</td>
<td>22</td>
</tr>
<tr>
<td>commented_code_linter</td>
<td>23</td>
</tr>
<tr>
<td>R topics documented:</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>---</td>
</tr>
<tr>
<td>common_mistakes_linters</td>
<td>24</td>
</tr>
<tr>
<td>condition_message_linter</td>
<td>25</td>
</tr>
<tr>
<td>configurable_linters</td>
<td>26</td>
</tr>
<tr>
<td>conjunct_test_linter</td>
<td>27</td>
</tr>
<tr>
<td>consecutive_assertion_linter</td>
<td>29</td>
</tr>
<tr>
<td>consistency_linters</td>
<td>30</td>
</tr>
<tr>
<td>correctness_linters</td>
<td>31</td>
</tr>
<tr>
<td>cyclocomp_linter</td>
<td>31</td>
</tr>
<tr>
<td>default_linters</td>
<td>32</td>
</tr>
<tr>
<td>default_settings</td>
<td>33</td>
</tr>
<tr>
<td>deprecated_linters</td>
<td>34</td>
</tr>
<tr>
<td>duplicate_argument_linter</td>
<td>35</td>
</tr>
<tr>
<td>efficiency_linters</td>
<td>36</td>
</tr>
<tr>
<td>empty_assignment_linter</td>
<td>37</td>
</tr>
<tr>
<td>equals_na_linter</td>
<td>38</td>
</tr>
<tr>
<td>exclude</td>
<td>39</td>
</tr>
<tr>
<td>executing_linters</td>
<td>40</td>
</tr>
<tr>
<td>expect_comparison_linter</td>
<td>40</td>
</tr>
<tr>
<td>expect_identical_linter</td>
<td>41</td>
</tr>
<tr>
<td>expect_length_linter</td>
<td>43</td>
</tr>
<tr>
<td>expect_lint</td>
<td>43</td>
</tr>
<tr>
<td>expect_lint_free</td>
<td>44</td>
</tr>
<tr>
<td>expect_named_linter</td>
<td>45</td>
</tr>
<tr>
<td>expect_not_linter</td>
<td>46</td>
</tr>
<tr>
<td>expect_null_linter</td>
<td>46</td>
</tr>
<tr>
<td>expect_s3_class_linter</td>
<td>47</td>
</tr>
<tr>
<td>expect_s4_class_linter</td>
<td>48</td>
</tr>
<tr>
<td>expect_true_false_linter</td>
<td>49</td>
</tr>
<tr>
<td>expect_type_linter</td>
<td>50</td>
</tr>
<tr>
<td>extraction_operator_linter</td>
<td>51</td>
</tr>
<tr>
<td>fixed_regex_linter</td>
<td>52</td>
</tr>
<tr>
<td>for_loop_index_linter</td>
<td>53</td>
</tr>
<tr>
<td>function_argument_linter</td>
<td>54</td>
</tr>
<tr>
<td>function_left_parentheses_linter</td>
<td>55</td>
</tr>
<tr>
<td>function_return_linter</td>
<td>57</td>
</tr>
<tr>
<td>get_r_string</td>
<td>58</td>
</tr>
<tr>
<td>get_source_expressions</td>
<td>59</td>
</tr>
<tr>
<td>ids_with_token</td>
<td>60</td>
</tr>
<tr>
<td>ifelse_censor_linter</td>
<td>61</td>
</tr>
<tr>
<td>if_not_else_linter</td>
<td>62</td>
</tr>
<tr>
<td>implicit_assignment_linter</td>
<td>63</td>
</tr>
<tr>
<td>implicit_integer_linter</td>
<td>65</td>
</tr>
<tr>
<td>indentation_linter</td>
<td>66</td>
</tr>
<tr>
<td>infix_spaces_linter</td>
<td>69</td>
</tr>
<tr>
<td>inner_combine_linter</td>
<td>70</td>
</tr>
<tr>
<td>is_lint_level</td>
<td>71</td>
</tr>
<tr>
<td>is_numeric_linter</td>
<td>72</td>
</tr>
<tr>
<td>keyword_quote_linter</td>
<td>73</td>
</tr>
<tr>
<td>lengths_linter</td>
<td>74</td>
</tr>
<tr>
<td>length_levels_linter</td>
<td>75</td>
</tr>
<tr>
<td>length_test_linter</td>
<td>75</td>
</tr>
<tr>
<td>library_call_linter</td>
<td>76</td>
</tr>
<tr>
<td>R topics documented:</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>line_length_linter</td>
<td>77</td>
</tr>
<tr>
<td>lint</td>
<td>78</td>
</tr>
<tr>
<td>lint-s3</td>
<td>80</td>
</tr>
<tr>
<td>Linter</td>
<td>81</td>
</tr>
<tr>
<td>linters</td>
<td>81</td>
</tr>
<tr>
<td>linters_with_defaults</td>
<td>85</td>
</tr>
<tr>
<td>linters_with_tags</td>
<td>86</td>
</tr>
<tr>
<td>literal_coercion_linter</td>
<td>87</td>
</tr>
<tr>
<td>make_linter_from_xpath</td>
<td>88</td>
</tr>
<tr>
<td>matrix_apply_linter</td>
<td>89</td>
</tr>
<tr>
<td>missing_argument_linter</td>
<td>89</td>
</tr>
<tr>
<td>missing_package_linter</td>
<td>90</td>
</tr>
<tr>
<td>modify_defaults</td>
<td>91</td>
</tr>
<tr>
<td>namespace_linter</td>
<td>92</td>
</tr>
<tr>
<td>nested_ifelse_linter</td>
<td>93</td>
</tr>
<tr>
<td>nonportable_path_linter</td>
<td>95</td>
</tr>
<tr>
<td>numeric_leading_zero_linter</td>
<td>95</td>
</tr>
<tr>
<td>object_length_linter</td>
<td>96</td>
</tr>
<tr>
<td>object_name_linter</td>
<td>97</td>
</tr>
<tr>
<td>object_usage_linter</td>
<td>99</td>
</tr>
<tr>
<td>outer_negation_linter</td>
<td>100</td>
</tr>
<tr>
<td>package_development_linters</td>
<td>101</td>
</tr>
<tr>
<td>package_hooks_linter</td>
<td>102</td>
</tr>
<tr>
<td>paren_body_linter</td>
<td>103</td>
</tr>
<tr>
<td>parse_exclusions</td>
<td>104</td>
</tr>
<tr>
<td>paste_linter</td>
<td>105</td>
</tr>
<tr>
<td>pipe_call_linter</td>
<td>107</td>
</tr>
<tr>
<td>pipe_consistency_linter</td>
<td>108</td>
</tr>
<tr>
<td>pipe_continuation_linter</td>
<td>109</td>
</tr>
<tr>
<td>pkg_testthat_linters</td>
<td>110</td>
</tr>
<tr>
<td>quotes_linter</td>
<td>111</td>
</tr>
<tr>
<td>readability_linters</td>
<td>112</td>
</tr>
<tr>
<td>read_settings</td>
<td>113</td>
</tr>
<tr>
<td>redundant_equals_linter</td>
<td>114</td>
</tr>
<tr>
<td>redundant_ifelse_linter</td>
<td>115</td>
</tr>
<tr>
<td>regex_subset_linter</td>
<td>116</td>
</tr>
<tr>
<td>repeat_linter</td>
<td>117</td>
</tr>
<tr>
<td>robustness_linters</td>
<td>118</td>
</tr>
<tr>
<td>routine_registration_linter</td>
<td>119</td>
</tr>
<tr>
<td>sarif_output</td>
<td>120</td>
</tr>
<tr>
<td>scalar_in_linter</td>
<td>120</td>
</tr>
<tr>
<td>semicolon_linter</td>
<td>121</td>
</tr>
<tr>
<td>seq_linter</td>
<td>122</td>
</tr>
<tr>
<td>sort_linter</td>
<td>123</td>
</tr>
<tr>
<td>spaces_inside_linter</td>
<td>125</td>
</tr>
<tr>
<td>spaces_left_parentheses_linter</td>
<td>126</td>
</tr>
<tr>
<td>sprintf_linter</td>
<td>126</td>
</tr>
<tr>
<td>strings_as_factors_linter</td>
<td>127</td>
</tr>
<tr>
<td>string_boundary_linter</td>
<td>128</td>
</tr>
<tr>
<td>style_linters</td>
<td>130</td>
</tr>
<tr>
<td>system_file_linter</td>
<td>131</td>
</tr>
<tr>
<td>todo_comment_linter</td>
<td>132</td>
</tr>
</tbody>
</table>
**absolute_path_linter**

Absolute path linter

---

**Description**

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

**Usage**

```r
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

# Following examples use raw character constant syntax introduced in R 4.0.

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

# okay
lint(
  text = 'R"./blah"',
  linters = absolute_path_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

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

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.

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())
Default undesirable functions and operators

Description

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

Usage

all_undesirable_functions

default_undesirable_functions

all_undesirable_operators

default_undesirable_operators

Format

A named list of character strings.

Details

The following functions are sometimes regarded as undesirable:

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

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

The following operators are sometimes regarded as undesirable:

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

---

**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**

```r
any_duplicated_linter()
```
any_is_na_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()
)

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

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

any_is_na_linter

Description

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 <- is always used for assignment.

Usage

assignment_linter(
  allow_cascading_assign = TRUE,
  allow_right_assign = FALSE,
  allow_trailing = TRUE,
  allow_pipe_assign = FALSE
)
Arguments

allow_cascading_assign
Logical, default TRUE. If FALSE, <<- and ->> are not allowed.

allow_right_assign
Logical, default FALSE. If TRUE, -> and <<- are allowed.

allow_trailing
Logical, default TRUE. If FALSE then assignments aren’t allowed at end of lines.

allow_pipe_assign
Logical, default FALSE. If TRUE, magrittr’s %<>% assignment is allowed.

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()
)

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

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

# customizing using arguments
code_lines <- "1 -> x\n2 ->> y"
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 except_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.
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

```r
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  

Description

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

Usage

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

Arguments

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

Tags

- configurable, package_development, robustness

See Also

- `linter` for a complete list of linters available in lintr.

Examples

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

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

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

lint(
  text = "str2lang(x)",
  linters = backport_linter("4.0.0")
)
```

Best practices linters

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

Linters

The following linters are tagged with `"best_practices"`:

- `absolutely_path_linter`
- `any_duplicated_linter`
best_practices_linters

- any_is_na_linter
- boolean_arithmetic_linter
- class_equals_linter
- commented_code_linter
- condition_message_linter
- conjunct_test_linter
- cyclocomp_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
- extraction_operator_linter
- fixed_regex_linter
- for_loop_index_linter
- function_argument_linter
- function_return_linter
- ifelse_censor_linter
- implicit_assignment_linter
- implicit integer_linter
- is_numeric_linter
- length_levels_linter
- lengths_linter
- library_call_linter
- literal_coercion_linter
- nonportable_path_linter
- outer_negation_linter
- paste_linter
- redundant_equals_linter
- redundant_ifelse_linter
- regex_subset_linter
- routine_registration_linter
- scalar_in_linter
- seq_linter
- sort_linter
boolean_arithmetic_linter

- system_file_linter
- T_and_F_symbol_linter
- undesirable_function_linter
- undesirable_operator_linter
- unnecessary_lambda_linter
- unnecessary_nested_if_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**

\[ \text{length}(\text{which}(x == y)) == 0 \] is the same as \( \text{!any}(x == y) \), 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**

```r
# 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()
)
```
brace_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)

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

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

Tags
configurable, default, readability, style

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

**Examples**

```r
# 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}"

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

---

**Description**

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

**Usage**

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

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lints</td>
<td>the linting results.</td>
</tr>
<tr>
<td>filename</td>
<td>the name of the output report</td>
</tr>
</tbody>
</table>
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

---

**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.

---

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](https://style.tidyverse.org/syntax.html#commas)
Examples

```r
# 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**

**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 mistake linters

Common mistake linters

Description

Linters highlighting common mistakes, such as duplicate arguments.
Linters

The following linters are tagged with 'common_mistakes':

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

See Also

`linters` for a complete list of linters available in lintr.

---

**condition_message_linter**

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

Description

This linter discourages combining condition functions like `stop()` with string concatenation functions `paste()` and `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

```r
# 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
- conjunct_test_linter
- cyclocomp_linter
- duplicate_argument_linter
- fixed_regex_linter
- if_not_else_linter
conjunct_test_linter

- implicit_assignment_linter
- implicit_integer_linter
- indentation_linter
- infix_spaces_linter
- library_call_linter
- line_length_linter
- missing_argument_linter
- namespace_linter
- nonportable_path_linter
- object_length_linter
- object_name_linter
- object_usage_linter
- paste_linter
- pipe_consistency_linter
- quotes_linter
- redundant_ifelse_linter
- semicolon_linter
- string_boundary_linter
- todo_comment_linter
- trailing_whitespace_linter
- undesirable_function_linter
- undesirable_operator_linter
- unnecessary_concatenation_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**

```r
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 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()
)
consecutive_assertion_linter

Force consecutive calls to assertions into just one when possible

Description

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

Usage

\begin{verbatim}
consecutive_assertion_linter()
\end{verbatim}

Tags

consistency, readability, style

See Also

\texttt{linters} for a complete list of linters available in lintr.

Examples

\begin{verbatim}
# will produce lints
lint(  
  text = "stopifnot(x); stopifnot(y)",
  linters = consecutive_assertion_linter()
)
\end{verbatim}
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()
)

### consistency_linters

**Consistency linters**

**Description**

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

**Linters**

The following linters are tagged with `consistency`:

- assignment_linter
- class_equals_linter
- condition_message_linter
- consecutive Assertion_linter
- function_argument_linter
- if_not_else_linter
- implicit integer_linter
- inner combine_linter
- is_numeric_linter
- keyword_quote_linter
- length_levels_linter
- literal_coercion_linter
- numeric_leading zero_linter
- object_name_linter
- paste_linter
- quotes_linter
- redundant ifelse_linter
**correctness_linters**

- scalar_in_linter
- seq_linter
- system_file_linter
- T_and_F_symbol_linter
- whitespace_linter

**See Also**

linters for a complete list of linters available in lintr.

---

**cyclocomp_linter**  
*Cyclomatic complexity linter*

**Description**

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

**Usage**

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

**Arguments**

- `complexity_limit`
  
  Maximum cyclomatic complexity, default 15. Expressions more complex than this are linted. See `cyclocomp::cyclocomp()`.
default_linters

Tags

best_practices, configurable, default, 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  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 25.

Linters

The following linters are tagged with 'default':

- assignment_linter
- brace_linter
- commas_linter
- commented_code_linter
- cyclocomp_linter
- equals_na_linter
default_settings

- 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_continuation_linter
- quotes_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
- comment_token: a GitHub token character
- comment_bot: decides if lintr comment bot on GitHub can comment on commits
- 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 13.

See Also

`read_settings()`, `default_linters`

Examples

```r
# 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**  
**Deprecation Notice**

Description

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

Linters

The following linters are tagged with 'deprecated':

- `closed_curly_linter`
- `consecutive_stopifnot_linter`
- `no_tab_linter`
- `open_curly_linter`
- `paren_brace_linter`
- `semicolon_terminator_linter`
- `single_quotes_linter`
- `unneeded_concatenation_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**

```r
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**

```r
# 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
- fixed_regex_linter
- ifelse_censor_linter
- inner_combine_linter
- length_test_linter
- lengths_linter
- literal_coercion_linter
- matrix_apply_linter
- nested_ifelse_linter
- outer_negation_linter
- redundant_equals_linter
- redundant_ifelse_linter
- regex_subset_linter
- routine_registration_linter
- scalar_in_linter
- seq_linter
- sort_linter
- string_boundary_linter
- undesirable_function_linter
- undesirable_operator_linter
- unnecessary_concatenation_linter
- unnecessary_lambda_linter
- vector_logic_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

documentation, best_practices, readability

See Also

linters for a complete list of linters available in lintr.

Examples

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

writeLines("x = {
")
lint(
  text = "x = {
",
  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 \( x == \text{NA} \), \( x != \text{NA} \) and \( x \%in\% \text{NA} \). Such usage is almost surely incorrect – checks for missing values should be done with `is.na()`.

Usage

```r
equals_na_linter()
```

Tags
```
common_mistakes, correctness, default, robustness
```

See Also
`linters` for a complete list of linters available in lintr.

Examples

```r
# 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()
)
```
**exclude**

Exclude lines or files from linting

**Description**

Exclude lines or files from linting

**Usage**

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

**Arguments**

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

**Details**

Exclusions can be specified in three different ways.

1. Single line in the source file. default: `# nolint`, possibly followed by a listing of linters to exclude. If the listing is missing, all linters are excluded on that line. The default listing format is `# nolint: linter_name, linter2_name`. There may not be anything between the colon and the line exclusion tag and the listing must be terminated with a full stop (.) for the linter list to be respected.

2. Line range in the source file. default: `# nolint start`, `# nolint end`. `# nolint start` accepts linter lists in the same form as `# nolint`.

3. Exclusions parameter, a list with named and/or unnamed entries. Outer elements have the following characteristics:

   (a) Unnamed elements specify filenames or directories.

   (b) Named elements are a vector or list of line numbers, with `Inf` indicating ’all lines’. The name gives a path relative to the config.

      i. Unnamed elements denote exclusion of all linters in the given path or directory.

      ii. Named elements, where the name specifies a linter, denote exclusion for that linter. For convenience, a vector can be used in place of a list whenever it would not introduce ambiguity, e.g. a character vector of files to exclude or a vector of lines to exclude.
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_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**

```r
# 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()
)
```

---

**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

```r
# 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()
)
```
Description

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

Usage

\begin{verbatim}
expect_length_linter()
\end{verbatim}

Tags

\texttt{best_practices, package_development, pkg_testthat, readability}

See Also

\texttt{linters} for a complete list of linters available in lintr.

Examples

\begin{verbatim}
# 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()
)
\end{verbatim}

Description

This is an expectation function to test that the lints produced by \texttt{lint} satisfy a number of checks.

Usage

\begin{verbatim}
expect_lint(content, checks, ..., file = NULL, language = "en")
\end{verbatim}
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.

Value

NULL, invisibly.

Examples

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

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

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

---

**expect_lint_free**

Test that the package is lint free

Description

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

Arguments
... arguments passed to lint_package()

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

Description

\texttt{testthat::expect_false()} exists specifically for testing that an output is \texttt{FALSE}. \texttt{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 \texttt{expect_false(A)} instead of \texttt{expect_true(!A)}.

Usage

\begin{verbatim}
expect_not_linter()
\end{verbatim}

Tags

\begin{verbatim}
best_practices, package_development, pkg_testthat, readability
\end{verbatim}

See Also

\begin{verbatim}
linters for a complete list of linters available in lintr.
\end{verbatim}

Examples

\begin{verbatim}
# will produce lints
lint(
    text = "expect_true(!x)",
    linters = expect_not_linter()
)

# okay
lint(
    text = "expect_false(x)",
    linters = expect_not_linter()
)
\end{verbatim}

expect_null_linter

Description

Require usage of \texttt{expect_null(x)} over \texttt{expect_equal(x, NULL)} and similar usages.

Usage

\begin{verbatim}
expect_null_linter()
\end{verbatim}

Details

\texttt{testthat::expect_null()} exists specifically for testing for \texttt{NULL} objects. \texttt{testthat::expect_equal()}, \texttt{testthat::expect_identical()}, and \texttt{testthat::expect_true()} can also be used for such tests, but it is better to use the tailored function instead.
**expect_s3_class_linter**

Tags

best_practices, package_development, pkg_testthat

See Also

linters for a complete list of linters available in lintr.

Examples

```r
# 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()
)
```

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
expect_s4_class_linter

See Also

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

Examples

```r
# 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

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

```r
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**

```r
# 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()
)
```

---

**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**

```r
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**

```r
# 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()
)```
expect_type_linter

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

expect_type_linter

<table>
<thead>
<tr>
<th>Require usage of</th>
<th>expect_type(x, type) over</th>
</tr>
</thead>
<tbody>
<tr>
<td>expect_equal(typeof(x), type)</td>
<td></td>
</tr>
</tbody>
</table>

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()
)
Description

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

Usage

```r
extraction_operator_linter()
```

Details

There are three subsetting operators in R (`[[`, `[]`, and `$`) and they interact differently with different data structures (atomic vector, list, data frame, etc.).

Here are a few reasons to prefer the `[[` operator over `[` or `$` when you want to extract an element from a data frame or a list:

- Subsetting a list with `[]` always returns a smaller list, while `[[` returns the list element.
- Subsetting a named atomic vector with `[]` returns a named vector, while `[[` returns the vector element.
- Subsetting a data frame (but not tibble) with `[]` is type unstable; it can return a vector or a data frame. `[[`, on the other hand, always returns a vector.
- For a data frame (but not tibble), `$` does partial matching (e.g. `df$a` will subset `df$abc`), which can be a source of bugs. `[[` doesn’t do partial matching.

For data frames (and tibbles), irrespective of the size, the `[[` operator is slower than `$`. For lists, however, the reverse is true.

Tags

`best_practices`, `style`

References

- Subsetting chapter from *Advanced R* (Wickham, 2019).

See Also

`linters` for a complete list of linters available in lintr.

Examples

```r
# will produce lints
lint(
  text = 'iris$Species',
  linters = extraction_operator_linter()
)
```

```r
lint(
```
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`

See Also

`lintr` for a complete list of linters available in `lintr`.

Examples

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

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

```r
# 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

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

```r
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](https://style.tidyverse.org/syntax.html#parentheses)
- `spaces_left_parentheses_linter()`

Examples

```r
# 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**

```r
function_return_linter()
```

**Tags**

`best_practices, readability`

**See Also**

`linters` for a complete list of linters available in lintr.

**Examples**

```r
# 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,
```
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 valid since R 4.0, 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 <- withr::local_tempfile(lines = "c('a', 'b')")
extr_xml <- get_source_expressions(tmp)$expressions[[1L]]$xml_parsed_content
writeLines(as.character(expr_as_xml))
get_r_string(expr_as_xml, "expr[2]") # "a"
get_r_string(expr_as_xml, "expr[3]") # "b"

# more importantly, extract strings under R>=4 raw strings

tmp4.0 <- withr::local_tempfile(lines = "c(R'(a\b)', R'--[a\"\"\"\"\"\b]--')")
extr_xml4.0 <- get_source_expressions(tmp4.0)$expressions[[1L]]$xml_parsed_content
**get_source_expressions**

**Parsed source file from a filename**

**Description**

This object is given as input to each linter.

**Usage**

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

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

Get parsed IDs by token

Description

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

Usage

```r
ids_with_token(source_expression, value, fun = `==`, source_file = NULL)
with_id(source_expression, id, source_file)
```

Arguments

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

Examples

```r
tmp <- withr::local_tempfile(lines = c("x <- 1", "y <- x + 1"))
get_source_expressions(tmp)
```
**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**

```r
tmp <- withr::local_tempfile(lines = c("x <- 1", "y <- x + 1"))
source_exprs <- get_source_expressions(tmp)
ids_with_token(source_exprs$expressions[[1L]], value = "SYMBOL")
with_id(source_exprs$expressions[[1L]], 2L)
```

---

**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.
### if_not_else_linter

Block statements like `if (!A) x else y` are equivalent to `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

```r
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

```r
# 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()
)
```

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. `capture.output()`).

**Usage**

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

**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](https://style.tidyverse.org/syntax.html#assignment)

**Examples**

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

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

```r
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(allowScoped = TRUE)
)

---

**implicit_integer_linter**

*Implicit integer linter*

**Description**

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

**Usage**

```r
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**

```r
# 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**

```r
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.

```r
# 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:

```r
# 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

customizable, 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) {
1 + 1
}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = indentation_linter()
)

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

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

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

# okay
code_lines <- "map(x, f,
  additional_arg = 42\n)"
writeLines(code_lines)
lint(}
infix_spaces_linter

```r
text = code_lines,
      linters = indentation_linter()
    )

code_lines <- "if (TRUE) {
  1 + 1
}"
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](https://style.tidyverse.org/syntax.html#infix-operators).

**Usage**

```r
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**

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

**Usage**

inner_combine_linter()

**Tags**

consistency, efficiency, readability

**See Also**

linters for a complete list of linters available in lintr.

**Examples**

```r
# 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

**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 <- withr::local_tempfile(lines = c("x <- 1", "y <- x + 1"))
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")

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**

```r
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**

```r
# 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 = 'class(z) %in% c("numeric", "integer", "factor")',
  linters = is_numeric_linter()
)
```
lengths_linter

Description

`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  \textit{Require usage of nlevels over length(levels(.))}

\textbf{Description}

\texttt{length(levels(x))} is the same as \texttt{nlevels(x)}, but harder to read.

\textbf{Usage}

\texttt{length_levels_linter()}

\textbf{Tags}

\texttt{best\_practices, consistency, readability}

\textbf{See Also}

\texttt{linters} for a complete list of linters available in lintr.

\textbf{Examples}

\begin{verbatim}
# 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()
)
\end{verbatim}

length_test_linter  \textit{Check for a common mistake where length is applied in the wrong place}

\textbf{Description}

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

length_test_linter()

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()
)

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

library_call_linter  Library call linter

Description

Force library calls to all be at the top of the script.

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.

Tags

best_practices, configurable, readability, style

See Also

linters for a complete list of linters available in lintr.
Examples

```r
# will produce lints
lint(
  text = "
    library(dplyr)
    print('test')
    library(tidyr)
    
    linters = library_call_linter()
  )

lint(
  text = "
    library(dplyr)
    print('test')
    library(tidyr)
    library(purrr)
    
    linters = library_call_linter()
  )

# okay
lint(
  text = "
    library(dplyr)
    print('test')
    
    linters = library_call_linter()
  )

lint(
  text = "
    # comment
    library(dplyr)
    
    linters = library_call_linter()
  )
```

line_length_linter

Description

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

Usage

`line_length_linter(length = 80L)`

Arguments

- `length` maximum line length allowed. Default is 80L (Hollerith limit).
### 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](https://style.tidyverse.org/syntax.html#long-lines)

### Examples

#### will produce lints

```r
lint(
  text = strrep("x", 23L),
  linters = line_length_linter(length = 20L)
)
```

#### okay

```r
lint(
  text = strrep("x", 21L),
  linters = line_length_linter(length = 40L)
)
```

---

### 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

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

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

    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, default TRUE. Whether to try and parse the settings; otherwise, the default_settings() are used.

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.


show_progress Logical controlling whether to show linting progress with a simple text progress bar via `utils::txtProgressbar()`. 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 unparsable 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

```r
t <- withr::local_tempfile(lines = "a=1", fileext = "R")
lint(f)  # linting a file
lint("a = 123\n")  # linting inline-code
lint(text = "a = 123")  # linting inline-code

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

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

Arguments

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

<table>
<thead>
<tr>
<th>type</th>
<th>type of lint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>message used to describe the lint error</td>
</tr>
<tr>
<td>line</td>
<td>code source where the lint occurred</td>
</tr>
<tr>
<td>ranges</td>
<td>a list of ranges on the line that should be emphasized.</td>
</tr>
<tr>
<td>linter</td>
<td>deprecated. No longer used.</td>
</tr>
</tbody>
</table>

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())

Arguments

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

Value

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

linters Available linters

Description

A variety of linters 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 (53 linters)
- common_mistakes (8 linters)
- configurable (34 linters)
- consistency (22 linters)
- correctness (7 linters)
- default (25 linters)
- deprecated (8 linters)
- efficiency (25 linters)
- executing (5 linters)
- package_development (14 linters)
- pkg_testthat (12 linters)
- readability (54 linters)
- robustness (14 linters)
- style (38 linters)

Linters

The following linters exist:

- absolute_path_linter (tags: best_practices, configurable, robustness)
- any_duplicated_linter (tags: best_practices, efficiency)
- any_is_na_linter (tags: best_practices, efficiency)
- assignment_linter (tags: 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)
- commas_linter (tags: configurable, default, readability, style)
- commented_code_linter (tags: best_practices, default, readability, style)
- condition_message_linter (tags: best_practices, consistency)
- conjunct_test_linter (tags: best_practices, configurable, package_development, pkg_testthat, readability)
- consecutive_assertion_linter (tags: consistency, readability, style)
- cyclocomp_linter (tags: best_practices, configurable, default, readability, style)
- 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)

• **extraction_operator_linter** (tags: best_practices, style)

• **fixed_regex_linter** (tags: best_practices, configurable, efficiency, readability)

• **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)

• **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)

• **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)

• **nonportable_path_linter** (tags: best_practices, configurable, robustness)

• **numeric_leading_zero_linter** (tags: consistency, readability, style)

• **object_length_linter** (tags: configurable, default, executing, readability, style)

• **object_name_linter** (tags: configurable, consistency, default, executing, style)
• **object_usage_linter** (tags: configurable, correctness, default, executing, 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, readability, style)
• **pipe_continuation_linter** (tags: default, readability, style)
• **quotes_linter** (tags: configurable, 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)
• **repeat_linter** (tags: readability, style)
• **routine_registration_linter** (tags: best_practices, efficiency, robustness)
• **scalar_in_linter** (tags: best_practices, 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)
• **string_boundary_linter** (tags: configurable, efficiency, readability)
• **strings_as_factors_linter** (tags: robustness)
• **system_file_linter** (tags: best_practices, consistency, readability)
• **T_and_F_symbol_linter** (tags: best_practices, consistency, default, readability, robustness, style)
• **todo_comment_linter** (tags: configurable, style)
• **trailing_blank_lines_linter** (tags: default, style)
• **trailing_whitespace_linter** (tags: configurable, default, style)
• **undesirable_function_linter** (tags: best_practices, configurable, efficiency, robustness, style)
• **undesirable_operator_linter** (tags: best_practices, configurable, efficiency, robustness, style)
• **unnecessary_concatenation_linter** (tags: configurable, efficiency, readability, style)
• **unnecessary_lambda_linter** (tags: best_practices, efficiency, readability)
• **unnecessary_nested_if_linter** (tags: best_practices, readability)
• **unnecessary_placeholder_linter** (tags: best_practices, readability)
• **unreachable_code_linter** (tags: best_practices, readability)
• **unused_import_linter** (tags: best_practices, common_mistakes, configurable, executing)
• **vector_logic_linter** (tags: best_practices, default, efficiency)
• **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)

with_defaults(..., default = default_linters)

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

See Also
• linters_with_tags for basing off tags attached to linters, possibly across multiple packages.
• 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 <- withr::local_tempfile(lines = "my_slightly_long_variable_name <- 2.3", fileext = "R")

lint(f, linters = linters_with_defaults(line_length_linter = line_length_linter(120)))

# the default linter list with a different line length cutoff

my_linters <- linters_with_defaults(line_length_linter = line_length_linter(120))

# omit the argument name if you are just using different arguments

my_linters <- linters_with_defaults(defaults = my_linters, object_name_linter("camelCase"))

# remove assignment checks (with NULL), add absolute path checks

my_linters <- linters_with_defaults(
    defaults = my_linters,
    assignment_linter = NULL,
    absolute_path_linter()
  )

# 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

```r
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.

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

```r
# '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"))
}
```
literal_coercion_linter

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()  
)
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")
)

Arguments

xpath Character string, an XPath identifying R code to lint. 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.

Examples

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

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()
)

missing_argument_linter

Missing argument linter

Description

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

```r
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

```r
# 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)
)
```

Description

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

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().
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)
)

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:

```r
ifelse(
    x == "a",
    ..., ...
) with nested calls
```
Here are a few ways to avoid nesting and make the code more readable:

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

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

- Use `dplyr::case_match()`

```r
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)

```r
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**

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

```r
# okay
lint(  
  text = 'dplyr::case_when(x == "a" ~ 1L, x == "b" ~ 2L, TRUE ~ 3L)',
  linters = nested_ifelse_linter()
)
```
линт(
    текст = 'data.table::fcase(x == "a", 1L, x == "b", 2L, default = 3L)',
    линтер = nested_ifelse_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()

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()
object_length_linter

Tags

consistency, readability, style

See Also

linters for a complete list of linters available in lintr.

Examples

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

# okay
lint("x <- 0.1",
     linters = numeric_leading_zero_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. \texttt{as.data.frame.my\_class} has length 8.
- leading \texttt{.}, e.g. \texttt{.my\_hidden\_function} has length 18.
- "\%\%" for infix operators, e.g. \texttt{\%my\_op\%} has length 5.
- trailing \texttt{<-} for assignment functions, e.g. \texttt{my\_attr<-} has length 7.

Note that this behavior relies in part on having packages in your Imports available; see the detailed note in \texttt{object\_name\_linter()} for more details.

Tags

configurable, default, executing, readability, style

See Also

\texttt{linters} for a complete list of linters available in lintr.

Examples

\begin{verbatim}
# 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)
)
\end{verbatim}
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

lintr for a complete list of linters available in lintr.

Examples

```r
# 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")
)
```
object_usage_linter

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_usage_linter  Object usage linter

Description

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

Usage

object_usage_linter(interpret_glue = TRUE, skip_with = TRUE)

Arguments

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

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()
)

outer_negation_linter

Require usage of !any(x) over all(!x), !all(x) over any(!x)

Description

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

Usage

outer_negation_linter()
package_development_linters

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(), and .onDetach() namespace hooks that will cause R CMD check issues. See Writing R Extensions for details.

Usage
package_hooks_linter()

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

Tags
correctness, package_development, style

See Also

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

# 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()
)

---

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

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

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

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

Description

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

Usage

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

Arguments

- `file` R source file
- `exclude` Regular expression used to mark lines to exclude.
- `exclude_next` Regular expression used to mark lines immediately preceding excluded lines.
- `exclude_start` Regular expression used to mark the start of an excluded range.
- `exclude_end` Regular expression used to mark the end of an excluded range.
- `exclude_linter` Regular expression used to capture a list of to-be-excluded linters immediately following a `exclude` or `exclude_start` marker.
- `exclude_linter_sep` Regular expression used to split a linter list into individual linter names for exclusion.
- `lines` A character vector of the content lines of `file`.
- `linter_names` Names of active linters.
Value

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

```
paste_linter
```

Description

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

Usage

```r
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 "never", 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 "always", 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(..., "")`.

Details

1. Block usage of `paste()` with `sep = ""`. `paste0()` is a faster, more concise alternative.
2. Block usage of `paste()` or `paste0()` with `collapse = ", ", "`. `toString()` is a direct wrapper for this, and alternatives like `glue::glueCollapse()` 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 `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.
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")
)

# 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)
)
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

Description

Check that pipe operators are used consistently by file, or optionally specify one valid pipe operator.

Usage

\[
\text{pipe\_consistency\_linter}(\text{pipe} = \text{c("auto", ")%>%", ")|>"})
\]

Arguments

\text{pipe} \quad \text{Which pipe operator is valid (either "%>%" or ")|>"). By default ("auto"), the linter has no preference but will check that each file uses only one type of pipe operator.}

Tags

configurable, readability, style

See Also

\text{linters} for a complete list of linters available in lintr.

Examples

\[
\begin{align*}
\text{# will produce lints} \\
\text{lint(} \\
\text{  text = "1:3 |> mean() %>% as.character()",} \\
\text{  linters = pipe\_consistency\_linter()} \\
\text{)} \\
\text{lint(} \\
\text{  text = "1:3 %>% mean() %>% as.character()",} \\
\text{  linters = pipe\_consistency\_linter("|>"))} \\
\text{# okay} \\
\text{lint(} \\
\text{  text = "1:3 %>% mean() %>% as.character()",} \\
\text{  linters = pipe\_consistency\_linter()} \\
\text{)} \\
\text{lint(} \\
\text{  text = "1:3 |> mean() |> as.character()",} \\
\text{  linters = pipe\_consistency\_linter()} \\
\text{)}
\end{align*}
\]
**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**

```r
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](https://style.tidyverse.org/pipes.html#long-lines-2)

**Examples**

```r
# 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()
)
```
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
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

customizable, consistency, default, readability, style

See Also

• linters for a complete list of linters available in lintr.

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
- commas_linter
- commented_code_linter
- conjunct_test_linter
- consecutive_assertion_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
- 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
- matrix_apply_linter
- nested_ifelse_linter
- numeric_leading_zero_linter
- object_length_linter
• object_usage_linter
• outer_negation_linter
• paren_body_linter
• pipe_call_linter
• pipe_consistency_linter
• pipe_continuation_linter
• quotes_linter
• redundant_equals_linter
• repeat_linter
• scalar_in_linter
• semicolon_linter
• sort_linter
• spaces_inside_linter
• spaces_left_parentheses_linter
• string_boundary_linter
• system_file_linter
• T_and_F_symbol_linter
• unnecessary_concatenation_linter
• unnecessary_lambda_linter
• unnecessary_nested_if_linter
• unnecessary_placeholder_linter
• unreachable_code_linter
• yoda_test_linter

See Also

linters for a complete list of linters available in lintr.

---

read_settings  Read lintr settings

Description

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

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

Usage

read_settings(filename)
### 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

```r
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

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

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

```r
# 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_subset_linter

Require usage of direct methods for subsetting strings via regex

Description

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

Usage

regex_subset_linter()

Details

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

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

Exceptions

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

Tags

best_practices, efficiency

See Also

`linters` for a complete list of linters available in lintr.
repeat_linter

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 = "grep(pattern, x, value = TRUE)",
    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.

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(
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`
• `backport_linter`
• `class_equals_linter`
• `equals_na_linter`
• `for_loop_index_linter`
• `missing_package_linter`
• `namespace_linter`
• `nonportable_path_linter`
• `routine_registration_linter`
• `seq_linter`
• `strings_as_factors_linter`
• `T_and_F_symbol_linter`
• `undesirable_function_linter`
• `undesirable_operator_linter`

See Also
`linters` for a complete list of linters available in lintr.
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()
)
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. %chin% from \{data.table\} is matched as well.

Usage

scalar_in_linter()

Details

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

Tags

best_practices, consistency, efficiency, readability

See Also

lintr for a complete list of linters available in lintr.
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;",
   )
```r
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() {
  a <- 1
  b <- 1}
"
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. It is safer to use `base::seq_len()` or `base::seq_along()` instead.

**Tags**

`best_practices, consistency, default, efficiency, robustness`

**See Also**

`lintr` for a complete list of linters available in lintr.
sort_linter

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()
)

# 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()
)

-----------
sort_linter Check for common mistakes around sorting vectors

Description

This linter checks for some common mistakes when using \texttt{order()} or \texttt{sort()}.

Usage

\texttt{sort_linter()}

Details

First, it requires usage of \texttt{sort()} over \texttt{.\texttt{order(.)}}. \texttt{\texttt{sort()}} is the dedicated option to sort a list or vector. It is more legible and around twice as fast as \texttt{.\texttt{order(.)}}, with the gap in performance growing with the vector size.

Second, it requires usage of \texttt{is.unsorted()} over equivalents using \texttt{sort()}. 
The base function \texttt{is.unsorted()} exists to test the sortedness of a vector. Prefer it to inefficient and less-readable equivalents like \texttt{x != sort(x)}. The same goes for checking \texttt{x == sort(x)} – use \texttt{!is.unsorted(x)} instead.

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

\textbf{Tags}

\texttt{best\_practices, efficiency, readability}

\textbf{See Also}

\texttt{linters} for a complete list of linters available in lintr.

\textbf{Examples}

# will produce lints
\begin{verbatim}
lint(
  text = "x[order(x)]",
  linters = sort_linter()
)
\end{verbatim}

\begin{verbatim}
lint(
  text = "x[order(x, decreasing = TRUE)]",
  linters = sort_linter()
)
\end{verbatim}

\begin{verbatim}
lint(
  text = "sort(x) == x",
  linters = sort_linter()
)
\end{verbatim}

# okay
\begin{verbatim}
lint(
  text = "x[sample(order(x))]",
  linters = sort_linter()
)
\end{verbatim}

\begin{verbatim}
lint(
  text = "y[order(x)]",
  linters = sort_linter()
)
\end{verbatim}

\begin{verbatim}
lint(
  text = "sort(x, decreasing = TRUE) == x",
  linters = sort_linter()
)
\end{verbatim}

# If you are sorting several objects based on the order of one of them, such # as:
\begin{verbatim}
x <- sample(1:26)
y <- letters
newx <- x[order(x)]
newy <- y[order(x)]
\end{verbatim}

# 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

ggettextf() calls are also included, since gettextf() is a thin wrapper around sprintf().

Tags

common_mistakes, correctness

See Also

liners 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()
)

# 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()
)

Description

Identify cases where stringsAsFactors should be supplied explicitly

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/.
string_boundary_linter

**Tags**

robustness

**See Also**

`linters` for a complete list of linters available in lintr.

**Examples**

```r
# 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**

`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 `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`

See Also

`linters` for a complete list of linters available in lintr.

Examples

```r
# 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)
)
```
Description

Linters highlighting code style issues.

Linters

The following linters are tagged with 'style':

- assignment_linter
- brace_linter
- commas_linter
- commented_code_linter
- consecutive_assertion_linter
- cyclocomp_linter
- extraction_operator_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
- package_hooks_linter
- paren_body_linter
- pipe_call_linter
- pipe_consistency_linter
- pipe_continuation_linter
- quotes_linter
- repeat_linter
- semicolon_linter
- spaces_inside_linter
- spaces_left_parentheses_linter
- T_and_F_symbol_linter
system_file_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()
)
todo_comment_linter  TODO comment linter

Description

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

Usage

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

Arguments

todo  Vector of strings that identify TODO comments.

Tags

configurable, style

See Also

linters for a complete list of linters available in lintr.

Examples

# will produce lints
lint(
  text = "x + y # TODO",
  linters = todo_comment_linter()
)

lint(
  text = "pi <- 1.0 # FIXME",
  linters = todo_comment_linter()
)

lint(
  text = "x <- TRUE # hack",
  linters = todo_comment_linter(todo = c("todo", "fixme", "hack"))
)

# 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()
)
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 <- withr::local_tempfile(lines = "x <- 1\n")
readLines(f)
lint(
  filename = f,
  linters = trailing_blank_lines_linter()
)

# okay
f <- withr::local_tempfile(lines = "x <- 1")
readLines(f)
lint(
  filename = f,
  linters = trailing_blank_lines_linter()
)
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()
)
T_and_F_symbol_linter

```r
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**

Avoid the symbols T and F, and use `TRUE` and `FALSE` instead.

**Usage**

```r
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](https://style.tidyverse.org/syntax.html#logical-vectors)

**Examples**

```r
# 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 (e.g. `base::return()`, `base::options()`, or `base::sapply()`) and suggest an alternative.

Usage

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

Arguments

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

Tags

- `best_practices`, `configurable`, `efficiency`, `robustness`, `style`

See Also

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

Examples

```r
# 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()"))
)
undesirable_operator_linter

Undesirable operator linter

Description

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

Usage

`undesirable_operator_linter(op = default_undesirable_operators)`

Arguments

op

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

Tags

`best_practices, configurable, efficiency, robustness, style`

See Also

`lintr` 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.\")
)

# 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.\")
)

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.
unnecessary_lambda_linter

**Tags**

configurable, efficiency, readability, style

**See Also**

*lintr* 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_nested_if_linter

Avoid unnecessary nested if conditional statements

Description

Avoid unnecessary nested if conditional statements
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

Usage

unnecessary_nested_if_linter()

Tags

best_practices, readability

See Also

liners for a complete list of linters available in lintr.

Examples

# will produce lints
writeLines("if (x) { 
  if (y) { 
    return(1L) 
  } 
}")
lint(
  text = "if (x) { 
    if (y) { 
      return(1L) 
    } 
}",
  linters = unnecessary_nested_if_linter()
)

# okay
writeLines("if (x && y) { 
  return(1L) 
}")
lint(
  text = "if (x && y) { 
    return(1L) 
}",
  linters = unnecessary_nested_if_linter()
)

writeLines("if (x) { 
  y <- x + 1L
  if (y) { 
    return(1L) 
  } 
}")
lint(
  text = "if (x) { 
    y <- x + 1L
    if (y) { 
      return(1L) 
    } 
}",
  linters = unnecessary_nested_if_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()`

Tags

`best_practices, readability`

See Also

`linters` for a complete list of linters available in lintr.
Examples

```r
# will produce lints
code_lines <- "f <- function() {
  return(1 + 1)
  2 + 2}
"; writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "f <- if (FALSE) {
  2 + 2}
"; writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "f <- while (FALSE) {
  2 + 2}
"; writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

# okay

code_lines <- "f <- function() {
  return(1 + 1)
}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "f <- if (foo) {
  2 + 2}
"; writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)

code_lines <- "f <- while (foo) {
  2 + 2}"
writeLines(code_lines)
lint(
  text = code_lines,
  linters = unreachable_code_linter()
)
```

unused_import_linter

Check that imported packages are actually used

Description

Check that imported packages are actually used
unused_import_linter

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 loading 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,
use_lintr

linters = unused_import_linter(allow_ns_usage = TRUE)
)

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

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

Arguments
path          Path to project root, where a .lintr file should be created. If the .lintr file already exists, an error will be thrown.
type          What kind of configuration to create?

  • tidyverse creates a minimal lintr config, based on the default linters (linters_with_defaults()).
    These are suitable for following the tidyverse style guide.
  • full creates a lintr config using all available linters via linters_with_tags().

Value
Path to the generated configuration, invisibly.

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

Examples
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
gen_best_practices, 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()
)

# okay
lint(
  text = "if (TRUE && FALSE) 1",
  linters = vector_logic_linter()
)
whitespace_linter

```r
lint(
    text = "if (TRUE && (TRUE || FALSE)) 4",
    linters = vector_logic_linter()
)
```

---

**Whitespace linter**

**Description**

Check that the correct character is used for indentation.

**Usage**

```r
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://blog.codinghorror.com/death-to-the-space-infidels/](https://blog.codinghorror.com/death-to-the-space-infidels/)

**See Also**

linters for a complete list of linters available in lintr.

**Examples**

```r
# will produce lints
lint(
    text = "\tx",
    linters = whitespace_linter()
)
```

```r
# 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**

```r
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.
xp_call_name

Value

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

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

eq expr: An xml_node or xml_nodeset, e.g. from xml2::xml_find_all().

deq 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.

ceq 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")
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()
)