The Journal of Open Source Software

Static Code Analysis for R

Jim Hester [1], **Florent Angly** [2], **Michael Chirico** [3], **Russ Hyde** [4], **Ren Kun** [5], **Indrajeet Patil** [6], **and Alexander Rosenstock** [7]

1 Netflix **2** The University of Queensland **3** Google **4** Jumping Rivers **5** Unknown **6** Carl Zeiss AG, Germany **7** Mathematisches Institut der Heinrich-Heine-Universität Düsseldorf

DOI: 10.21105/joss.07240

Software

- Review
- Repository
- Archive

Editor: Oskar Laverny

Reviewers:

- @JosiahParry
- @SaranjeetKaur

Submitted: 03 September 2024
Published: 03 April 2025

Statement of Need

In computer programming, "linting" is the process of analyzing the source code to identify possible programming and stylistic problems (Wikipedia contributors, 2024a) and a linter is a tool used for linting. A linter analyzes code to identify potential errors, stylistic issues, or deviations from coding standards. It helps ensure consistency, readability, and best practices by flagging common mistakes, such as syntax errors, unused variables, or improper formatting. Linters are essential for improving code quality, preventing bugs, and maintaining a clean codebase, especially in collaborative development environments (Wikipedia contributors, 2024b). `{lintr}` is an open-source package that provides linters for the R programming language, which is an interpreted, dynamically-typed programming language (R Core Team, 2023), and is used by a wide range of researchers and data scientists. `{lintr}` can thus act as a valuable tool for R users to help improve the quality and reliability of their code.

Features

By default, `{lintr}` enforces the tidyverse style guide Müller et al. (2024). In this respect, it differs from other static code analysis tools in R (like `{codetools}` (Tierney, 2024)), which are not opinionated and don't enforce any particular style of writing code, but, rather, check R code for possible problems (incidentally, `{lintr}` uses `{codetools}` as a backend for object usage linters). Additionally, `{lintr}` is concerned only with R code, so code-adjacent text like inline `{roxygen2}` comments (Wickham et al., 2024) will not be covered (cf. `{roxylint}` (Kelkhoff, 2024)).

As of this writing, `{lintr}` offers 113 linters.

```
library(lintr)

length(all_linters())
#> [1] 113
```

Naturally, we can't discuss all of them here. To see the most up-to-date details about all the available linters, we encourage readers to visit https://lintr.r-lib.org/dev/reference/index.html#individual-linters.

We will showcase one linter for each kind of common problem found in R code.

- **Best practices**

`{lintr}` offers linters that can detect problematic antipatterns and suggest alternatives that follow best practices.

For example, 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. The

`redundant_ifelse_linter()` linter detects such discouraged usages.

```
lint(
  text = "ifelse(x >= 2.5, TRUE, FALSE)",
  linters = redundant_ifelse_linter()
)
#> <text>:1:1: warning: [redundant_ifelse_linter] Just use the
#>     logical condition (or its negation) directly instead of
#>     calling ifelse(x, TRUE, FALSE)
#> ifelse(x >= 2.5, TRUE, FALSE)
#> ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
lint(
  text = "x >= 2.5",
  linters = redundant_ifelse_linter()
)
#> i No lints found.
```

- **Efficiency**

Sometimes users might not be aware of a more efficient way offered by R for carrying out a computation. {lintr} offers linters to improve code efficiency by avoiding common inefficient patterns.

For example, the `any_is_na_linter()` linter detects usages of `any(is.na(x))` and suggests `anyNA(x)` as a more efficient alternative to detect presence of *any* missing values.

```
lint(
  text = "any(is.na(x), na.rm = TRUE)",
  linters = any_is_na_linter()
)
#> <text>:1:1: warning: [any_is_na_linter] anyNA(x) is better
#>     than any(is.na(x)).
#> any(is.na(x), na.rm = TRUE)
#> ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

`anyNA()` in R is more efficient than `any(is.na())` because it stops execution once a missing value is found, while `is.na()` evaluates the entire vector.

```
lint(
  text = "anyNA(x)",
  linters = any_is_na_linter()
)
#> i No lints found.
```

- **Readability**

Coders spend significantly more time reading than writing code (McConnell, 2004). Thus, writing readable code makes the code more maintainable and reduces the possibility of introducing bugs stemming from a poor understanding of the code.

{lintr} provides a number of linters that suggest more readable alternatives. For example, `comparison_negation_linter()` blocks usages like `!(x == y)` where a direct relational operator is appropriate.

```
lint(
  text = "!x == 2",
  linters = comparison_negation_linter()
)
#> <text>:1:1: warning: [comparison_negation_linter] Use x !=
#>     y, not !(x == y).
```

```
#> !x == 2
#> ^~~~~~
```

Note also the complicated operator precedence. The more readable alternative here uses `!=`:

```
lint(
  text = "x != 2",
  linters = comparison_negation_linter()
)
#> i No lints found.
```

- **Tidyverse style**

`{lintr}` also provides linters to enforce the style used throughout the `{tidyverse}` (Wickham et al., 2019) ecosystem of R packages. This style of coding has been outlined in the tidyverse style guide (Wickham, 2023).

For example, the style guide recommends using snake_case for identifiers:

```
lint(
  text = "MyVar <- 1L",
  linters = object_name_linter()
)
#> <text>:1:1: style: [object_name_linter] Variable and
#>     function name style should match snake_case or symbols.
#> MyVar <- 1L
#> ^~~~~
```

```
lint(
  text = "my_var <- 1L",
  linters = object_name_linter()
)
#> i No lints found.
```

- **Common mistakes**

One category of linters helps you detect some common mistakes statically and provide early feedback.

For example, duplicate arguments in function calls can sometimes cause run-time errors:

```
mean(x = 1:5, x = 2:3)
#> Error in mean(x = 1:5, x = 2:3): formal argument "x" matched by multiple actual argum
```

But `duplicate_argument_linter()` can check for this statically:

```
lint(
  text = "mean(x = 1:5, x = 2:3)",
  linters = duplicate_argument_linter()
)
#> <text>:1:15: warning: [duplicate_argument_linter] Avoid
#>     duplicate arguments in function calls.
#> mean(x = 1:5, x = 2:3)
#>               ^
```

Even for cases where duplicate arguments are not an error, this linter explicitly discourages duplicate arguments.

```
lint(
  text = "list(x = TRUE, x = FALSE)",
  linters = duplicate_argument_linter()
)
```

```
#> <text>:1:16: warning: [duplicate_argument_linter] Avoid
#>     duplicate arguments in function calls.
#> list(x = TRUE, x = FALSE)
#>                ^
```

This is because objects with duplicated names objects can be hard to work with programmatically and should typically be avoided.

```
l <- list(x = TRUE, x = FALSE)
l["x"]
#> $x
#> [1] TRUE
l[names(l) == "x"]
#> $x
#> [1] TRUE
#>
#> $x
#> [1] FALSE
```

Extensibility

{lintr} is designed for extensibility by allowing users to easily create custom linting rules. There are two main ways to customize it:

- Use additional arguments in existing linters. For example, although tidyverse style guide prefers snake_case for identifiers, if a project's conventions require it, the relevant linter can be customized to support it:

```
lint(
  text = "my.var <- 1L",
  linters = object_name_linter(styles = "dotted.case")
)
#> i No lints found.
```

- Create new linters (by leveraging functions like `lintr::make_linter_from_xpath()`) tailored to match project- or organization-specific coding standards.

Indeed, {goodpractice} (Padgham et al., 2024) bundles a set of custom linters that are not part of the default set of {lintr} linters, while {box.linters} (Basa & Nowicki, 2024) extends {lintr} to support {box} modules (Rudolph, 2024) and {checklist} includes linters as one of the strict checks for R packages (Onkelinx, 2024). {flir} (Bacher, 2024) is a Rust-backed analogue inspired by {lintr} that also provides support for fixing lints.

Benefits of using {lintr}

There are several benefits to using {lintr} to analyze and improve R code. One of the most obvious is that it can help users identify and fix problems in their code, which can save time and effort during the development process. By catching issues early on, {lintr} can help prevent bugs and other issues from creeping into code, which can save time and effort when it comes to debugging and testing.

Another benefit of {lintr} is that it can help users write more readable and maintainable code. By enforcing a consistent style and highlighting potential issues, {lintr} can help users write code that is easier to understand and work with. This is especially important for larger projects or teams, where multiple contributors may be working on the same codebase and it is important to ensure that code is easy to follow and understand, particularly when frequently switching context among code primarily authored by different people. {lintr} is designed to

be easy to use and integrate into existing workflows, and can be used as part of an automated build or continuous integration process. {lintr} also integrates with a number of popular IDEs and text editors, such as RStudio and Visual Studio Code, making it convenient for users to run {lintr} checks on their code as they work.

It can also be a useful tool for teaching and learning R. By providing feedback on code style and potential issues, it can help users learn good coding practices and improve their skills over time. This can be especially useful for beginners, who may not yet be familiar with all of the best practices for writing R code.

Finally, {lintr} has had a large and active user community since its birth in 2014 which has contributed to its rapid development, maintenance, and adoption. At the time of writing, {lintr} is in a mature and stable state and therefore provides a reliable API that is unlikely to feature fundamental breaking changes.

Conclusion

{lintr} is a valuable tool for R users to help improve the quality and reliability of their code. Its static code analysis capabilities, combined with its flexibility and ease of use, make it relevant and valuable for a wide range of applications.

Licensing and Availability

{lintr} is licensed under the MIT License, with all source code openly developed and stored on GitHub (https://github.com/r-lib/lintr), along with a corresponding issue tracker for bug reporting and feature enhancements.

Conflicts of interest

The authors declare no conflict of interest.

Funding

This work was not financially supported by any of the affiliated institutions of the authors.

Acknowledgments

{lintr} would not be possible without the immense work of the R-core team who maintain the R language and we are deeply indebted to them. We are also grateful to all contributors to {lintr}.

References

Bacher, E. (2024). *Flir: Find and fix lints in R code*. https://flir.etiennebacher.com

Basa, R. R., & Nowicki, J. (2024). *Box.linters: Linters for 'box' modules*. https://doi.org/10.32614/CRAN.package.box.linters

Kelkhoff, D. (2024). *Roxylint: Lint 'roxygen2'-generated documentation*. https://doi.org/10.32614/CRAN.package.roxylint

McConnell, S. (2004). *Code complete*. Pearson Education.

Müller, K., Walthert, L., & Patil, I. (2024). *Styler: Non-invasive pretty printing of r code*. https://doi.org/10.32614/CRAN.package.styler

Onkelinx, T. (2024). *Checklist: A thorough and strict set of checks for R packages and source code. Version 0.4.0*. https://doi.org/10.5281/zenodo.4028303

Padgham, M., Marks, K., de Bortoli, D., Csardi, G., Frick, H., Jones, O., & Alexander, H. (2024). *Goodpractice: Advice on r package building*. https://doi.org/10.32614/CRAN.package.goodpractice

R Core Team. (2023). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. https://www.R-project.org/

Rudolph, K. (2024). *Box: Write reusable, composable and modular R code*. https://doi.org/10.32614/CRAN.package.box

Tierney, L. (2024). *Codetools: Code analysis tools for r*. https://doi.org/10.32614/CRAN.package.codetools

Wickham, H. (2023). *The tidyverse style guide*. https://style.tidyverse.org/index.html

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., … Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686. https://doi.org/10.21105/joss.01686

Wickham, H., Danenberg, P., Csárdi, G., & Eugster, M. (2024). *roxygen2: In-line documentation for R*. https://doi.org/10.32614/CRAN.package.roxygen2

Wikipedia contributors. (2024a). *Lint (software) — Wikipedia, the free encyclopedia*. https://en.wikipedia.org/w/index.php?title=Lint_(software)&oldid=1260589258

Wikipedia contributors. (2024b). *Static program analysis — Wikipedia, the free encyclopedia*. https://en.wikipedia.org/w/index.php?title=Static_program_analysis&oldid=1218663830