Package 'roger'

October 20, 2021

Type Package

Title Automated Grading of R Scripts

Version 0.99-1

Date 2021-10-19

Description Tools for grading the coding style and documentation of R scripts. This is the R component of Roger the Omni Grader, an automated grading system for computer programming projects based on Unix shell scripts; see <https://gitlab.com/roger-project>. The package also provides an R interface to the shell scripts. Inspired by the lintr package.

Depends R (>= 4.0.0)

Imports utils

Suggests tinytest

SystemRequirements roger-base (for interface functions only)

License GPL (>= 2)

URL https://roger-project.gitlab.io

BugReports https://gitlab.com/roger-project/roger-rpkg/-/issues

Encoding UTF-8

NeedsCompilation no

Author Vincent Goulet [aut, cre],
Samuel Fréchette [aut],
Jean-Christophe Langlois [aut],
Jim Hester [ctb]

Maintainer Vincent Goulet <vincent.goulet@act.ulaval.ca>

Repository CRAN

Date/Publication 2021-10-20 04:40:02 UTC

R topics documented:

`assignment_style` *Validation of the Assignment Operator*

Description

Check that the left assign symbol (`<-`) is used to assign values to objects instead of the equation assign symbol (`=`).

Usage

```
assignment_style(srcData)
```

Arguments

`srcData`	a list as returned by `getSourceData`.

Details

Although `=` is a proper assignment symbol at the top-level, its use is discouraged in R. Good coding practices dictate to rather use `<-` for (left) assignment to objects.

Value

Boolean. When `FALSE`, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

`lines`	faulty line numbers;
`message`	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct coding practice
fil <- tempfile(fileext = ".R")
cat("x <- 2", file = fil, sep = "\n")
assignment_style(getSourceData(fil))

## Incorrect coding practice
cat("x = 2", file = fil, sep = "\n")
assignment_style(getSourceData(fil))
```

---

commas_style *Validation of Spacing Around Commas*

Description

Check that commas are never preceded by a space and always followed by one, unless the comma ends the line.

Usage

```
commas_style(srcData)
```

Arguments

srcData	a list as returned by getSourceData.

Details

Good coding practices dictate to follow commas by a space, unless it ends its line, and to never put a space before.

Value

Boolean. When FALSE, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

lines	faulty line numbers;
message	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct use of spacing around a comma
fil <- tempfile(fileext = ".R")
cat("x <- c(2, 3, 5)", "crossprod(2,", "x)",
    file = fil, sep = "\n")
commas_style(getSourceData(fil))

## Incorrect use of spacing around a comma
fil <- tempfile(fileext = ".R")
cat("x <- c(2,3, 5)", "crossprod(2 ,", "x)",
    file = fil, sep = "\n")
commas_style(getSourceData(fil))
```

`curly_braces_style` *Validation of the Positioning of Braces*

Description

Check that the opening and closing braces are positioned according to standard bracing styles rules.

Usage

```
close_brace_style(srcData)
open_brace_style(srcData, style = c("R", "1TBS"))
open_brace_unique_style(srcData)
```

Arguments

`srcData`	a list as returned by `getSourceData`.
`style`	character string of a supported bracing style.

Details

Good coding practices dictate to use one bracing style uniformly in a script.

The "R" bracing style has both opening and closing braces on their own lines, left aligned with their corresponding statement:

```
if (x > 0)
{
    ...
}
```

The "1TBS" bracing style, also known at "K&R" style, has the opening brace immediately follow its corresponding statement, separated by a space. The closing brace lies on its own line, left aligned with the statement:

```
if (x > 0) {
    ...
}
```

`open_brace_style` validates that the coding style in argument is used for opening braces in a script file. `open_brace_unique_style` validates that only one style is used throughout the script.

These functions use `getParseText` and, therefore, require that the `"keep.source"` option is `TRUE`.

Value

Boolean. When `FALSE`, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

`lines`	faulty line numbers;
`message`	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct positioning of braces in R bracing style
fil <- tempfile(fileext = ".R")
cat("x <- 2",
    "if (x <= 2)",
    "{",
    "    y <- 3",
    "    x + y",
    "}",
    file = fil, sep = "\n")
srcData <- getSourceData(fil)
open_brace_style(srcData, style = "R")
close_brace_style(srcData)

## Above code in invalid in 1TBS bracing style
open_brace_style(srcData, style = "1TBS")

## Incorrect positioning of the opening brace and
## misalignment of the closing brace
fil <- tempfile(fileext = ".R")
cat("f <- function(x) {",
    "    x^2",
    "  }",
    file = fil, sep = "\n")
srcData <- getSourceData(fil)
```

```
open_brace_style(srcData, style = "R")
close_brace_style(srcData)

## Incorrect simultaneous use of two bracing styles
fil <- tempfile(fileext = ".R")
cat("x <- 2",
    "if (x <= 2)",
    "{",
    "    y <- 3",
    "    x + y",
    "}",
    "for (i in 1:5) {",
    "    x + i",
    "}",
    file = fil, sep = "\n")
open_brace_unique_style(getSourceData(fil))
```

---

documentation_linters *Validation of Documentation*

Description

Check for proper documentation of a function in the comments of a script file, and if certain mandatory sections are present.

The expected documentation format is not unlike R help pages; see details.

Usage

```
any_doc(srcData, ...)

signature_doc(srcData, ...)

section_doc(srcData, pattern, ...)
description_section_doc(srcData, ...)
arguments_section_doc(srcData, ...)
value_section_doc(srcData, ...)
examples_section_doc(srcData, ...)

formals_doc(srcData, ...)
```

Arguments

srcData	a list as returned by getSourceData.
pattern	character string containing a regular expression describing a keyword to match in the documentation.
...	further arguments passed to grepl.

Details

These functions check the documentation provided with function definitions in a script file. Lines starting with at least one comment character # (in column 1) are considered documentation.

`any_doc` checks that the file contains at least some documentation.

`signature_doc` checks that the signature (or usage information) of every function is present in the documentation.

`section_doc` checks that the documentation contains a section title corresponding to `pattern` for every (or as many) function definition. Functions `description_section_doc`, `arguments_section_doc`, `value_section_doc` and `examples_section_doc` are wrappers for common patterns `Description`, `Arguments?`, `Value` and `Examples?`, respectively.

`formals_doc` checks that the description of every formal argument is present in the documentation.

Value

Boolean. When `FALSE`, a message indicates the nature of the error, and the returned value has the following attributes:

`nlines`	number of lines checked for documentation (`any_doc` only);
`fun`	names of functions without a signature in the documentation (`signature_doc` only);
`sections`	number of sections missing from in the documentation (`section_doc` and wrapper functions only);
`formals`	formal arguments missing from the documentation (`formals_doc` only);
`message`	text of the error message.

References

Goulet, V., Programmer avec R for the standard documentation format these functions were developed for.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Define script file with embedded documentation.
fil <- tempfile(fileext = ".R")
cat(file = fil, "
###
### foo(x, y = 2)
###
##  Adding two vectors
##
##  Arguments
##
##  x: a vector
##  y: another vector
```

```
##
##  Value
##
##  Sum of the two vectors.
##
##  Examples
##
##  foo(1:5)
##
foo <- function(x, y = 2)
    x + y
")
fooData <- getSourceData(fil)

## Elements present in the documentation
any_doc(fooData)
arguments_section_doc(fooData)
value_section_doc(fooData)
examples_section_doc(fooData)
formals_doc(fooData)

## Missing section title
description_section_doc(fooData)
```

`getSourceData` *Get Parse Information and Source Code*

Description

Get parse information and source code from an R script file.

Usage

```
getSourceData(file, encoding, keep.source = getOption("keep.source"))
```

Arguments

`file`	a connection object or a character.
`encoding`	encoding to be assumed for input strings.
`keep.source`	a logical value; if `TRUE`, keep source reference information.

Details

The parse information of the script file is obtained using `getParseData`. The source code is read in using `readLines`. Arguments `file`, `encoding` and `keep.source` should be compatible with these functions.

Linters using results of this function may not work properly if the `encoding` argument does not match the encoding of the script file.

Value

A list of two elements:

parseData	parse information data frame;
Lines	source code character vector of length the number of lines in the file.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

fil <- tempfile(fileext = ".R")
cat("## A simple function",
    "foo <- function(x, y)",
    "{",
    "    z <- x + y",
    "    if (z > 3)",
    "        (x * y)^2",
    "    else",
    "        sqrt(x * y)",
    "}",
    file = fil, sep = "\n")
getSourceData(fil)
```

---

line_length_style *Validation of Length of Code and Comments Lines*

Description

Check that the length of code and comment lines does not exceed a given limit in number of characters.

Usage

```
line_length_style(srcData, nchar_max = 80L)
```

Arguments

srcData	a list as returned by getSourceData.
nchar_max	maximum number of characters per line.

Details

Good coding practice dictates to limit line length to a value around 80 characters for readability purposes.

The line length is determined from the column number ending each line of code in the parse information. This may give unintended results if the encoding used in getParseData does not match the encoding of the script file and the latter contains non ASCII characters.

Value

Boolean. When FALSE, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

lines	faulty line numbers;
message	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Valid line length
fil <- tempfile(fileext = ".R")
cat("## innocuous comment",
    "x <- 2",
    "y <- 3",
    "x + y",
    file = fil, sep = "\n")
line_length_style(getSourceData(fil))

## Limit valid line length to only 20 characters
fil <- tempfile(fileext = ".R")
cat("## no longer innocuous comment",
    "x <- 2",
    "y <- 3",
    "x + y",
    file = fil, sep = "\n")
line_length_style(getSourceData(fil), 20)
```

---

nomagic_style *Validation of Magic Numbers*

Description

Check the absence of *magic numbers* in code.

Usage

```
nomagic_style(srcData, ignore = c(-1, 0, 1, 2, 100),
              ignore.also = NULL)
```

Arguments

srcData	a list as returned by getSourceData.
ignore	vector of numbers not considered as magic.
ignore.also	vector of numbers not considered as magic in addition to those in ignore.

Details

Good coding practices dictate to avoid using "magic numbers" (unnamed or insufficiently documented numerical constants) in code. For example, in the expression `y <-x + 42`, `42` is a magic number.

When scanning the code for magic numbers, the following numerical constants are ignored:

- numbers in vectors `ignore` and `ignore.also`, with or without a suffix `L` denoting integer values;
- numbers used as the only expression in indexing;
- numbers in "simple" assignments to variables with all uppercase names.

"Simple" assignments serve to assign magic numbers to objects. Such expressions should contain at most three levels of sub-expressions and hold on a single line of code. The following expressions contain one, two and three levels of sub-expressions, respectively: `MAX <-4294967295`, `MAX <-4294967296 -1`, `MAX <-2^32 -1`.

Value

Boolean. When `FALSE`, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

`lines`	faulty line numbers;
`message`	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

fil <- tempfile(fileext = ".R")
cat("MAX <- 2^6 - 1",
    "size <- 42",
    "x <- rnorm(MAX)",
    "runif(123)",
    "x[1]",
    "x[1] * 7 + 2",
    "x[33]",
    "x * 100",
    "x <- numeric(0)",
    "y <- logical(5)",
    file = fil, sep = "\n")

## Default list of ignored magic numbers
nomagic_style(getSourceData(fil))

## Additional exceptions
nomagic_style(getSourceData(fil), ignore.also = c(5, 42))
```

ops_spaces_style *Validation of Spacing Around Operators*

Description

Check that spacing around infix and unary operators is valid.

Usage

```
ops_spaces_style(srcData, ops = ops_list)
```

Arguments

srcData	a list as returned by getSourceData.
ops	vector of R parser tokens corresponding to infix operators to check for correct spacing; see details.

Details

Good coding practices dictate to surround infix operators with spaces (including line breaks) and to follow unary operators immediately by their argument.

The default value for argument ops is an internal object containing the following R parser tokens: '+', '-', '*', GT, GE, LT, LE, EQ, NE, AND, OR, AND2, OR2, '!', LEFT_ASSIGN, RIGHT_ASSIGN, EQ_ASSIGN, EQ_SUB, SPECIAL.

Value

Boolean. When FALSE, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

lines	faulty line numbers;
message	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct use of spacing around '+' and '-' operators
fil <- tempfile(fileext = ".R")
cat("2 + 3",
    "-2",
    "(4\n + 2)",
    "4 +\n2",
    file = fil, sep = "\n")
ops_spaces_style(getSourceData(fil), c("'+'", "'-'"))
```

```
## Incorrect use of spacing around '>' and '!' operators
fil <- tempfile(fileext = ".R")
cat("2> 3",
    "4 >2",
    "6>3",
    "! FALSE",
    "!\nFALSE",
    file = fil, sep = "\n")
ops_spaces_style(getSourceData(fil), c("GT", "'!'"))
```

`parentheses_style` *Validation of Spacing Around Parentheses*

Description

Check that spacing around parentheses is valid.

Usage

```
open_parenthesis_style(srcData)
close_parenthesis_style(srcData)
left_parenthesis_style(srcData)
```

Arguments

`srcData`	a list as returned by `getSourceData`.

Details

Good coding practices dictate the correct spacing around parentheses. First, opening parentheses should not be immediately followed by a space. Second, closing parentheses should not be immediately preceded by a space. Third, left (or opening) parentheses should always be preceded by a space, except in function calls, at the start of sub-expressions, after operators `/` and `^`, or after an optional left parenthesis.

Value

Boolean. When `FALSE`, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

`lines`	faulty line numbers;
`message`	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct use of spacing around parentheses
fil <- tempfile(fileext = ".R")
cat("x <- c(2, 3, 5)",
    "if (any((2 * x) > 4))",
    "    sum(x)",
    "1/(x + 1)",
    "2^(x - 1)",
    "2^((x - 1))",
    file = fil, sep = "\n")
srcData <- getSourceData(fil)
open_parenthesis_style(srcData)
close_parenthesis_style(srcData)
left_parenthesis_style(srcData)

## Incorrect use of spacing around parentheses
fil <- tempfile(fileext = ".R")
cat("x <- c(2, 3, 5 )",
    "if(any(x > 4))",
    "    sum( x )",
    file = fil, sep = "\n")
srcData <- getSourceData(fil)
open_parenthesis_style(srcData)
close_parenthesis_style(srcData)
left_parenthesis_style(srcData)
```

`roger-interface` *R Interface for Roger Command Line Tools*

Description

R interfaces to the Roger base system command line tools `roger checkreq`, `roger clone`, `roger grade`, `roger push` and `roger validate`.

Usage

```
checkreq(file = "./requirements.txt", ...)

clone(project, pattern, page_limit = NULL, machine = NULL,
      curl_options = NULL, api = "bitbucket", ...)

grade(dir, config_file = NULL, time_limit = NULL,
      output_file = NULL, ...)

push(repos, branch, add_file = NULL, create_branch = FALSE,
```

```
    file = NULL, message = NULL, ...)

validate(dir, config_file = NULL, check_local_repos = TRUE, ...)
```

Arguments

project	name of a Git project containing repositories.
pattern	regular expression pattern.
dir	character vector of directory names containing projects to grade or validate; only the first one is used by validate.
repos	character vector of Git repository names to publish grading results into.
branch	name of the branch in which to publish the grading results (identical for every repository).
add_file	character vector of file names to publish along with the grading results.
api	character string; name of the REST API used to retrieve the urls of the repositories. Currently only the BitBucket API is supported.
check_local_repos	boolean; check the status of the local repository?
config_file	name of grading or validation configuration file; overrides the defaults gradeconf and valideconf.
create_branch	boolean; is branch a new branch to create in the repositories?
curl_options	character vector of command line options to pass to curl.
file	requirements file name for checkreq; name of the grading results file for push (overriding the default, locale dependent, value).
machine	URI and context of the Git server.
message	character vector of commit messages pasted together to form a single paragraph.
output_file	grading results file name; if NULL or -, results are written to standard output.
page_limit	integer; value of the REST API parameter limit indicating the number of results to return per page.
time_limit	date and time in ISO 8601 format (YYYY-MM-DD HH:MM:SS) by which to grade a project in a Git repository.
...	further arguments passed to system2.

Details

These functions build calls to the Roger base system command line tools and execute them using system2.

Command line option values are always quoted with shQuote.

Refer to the command line tools documentation for detailed information on the arguments and options.

Value

Character vector containing the standard output and standard error of the command line tools.

Note

The interface functions require that the Roger base system is installed on your machine and in your system path. See the Roger Project download page https://roger-project.gitlab.io/download/.

Examples

```
## Sample usage for students
## Not run: ## Validate the project in the current directory using the
## configuration file 'validateconf-prototype'.
validate(".", config_file = "validateconf-prototype")
## End(Not run)

## Sample usage for graders.
## Not run: ## First check the availability of the grading tools.
checkreq()

## Clone all repositories in the BitBucket project 'a2020-12345'
## matching the pattern '[0-9]{9}_prototype'.
clone("a2042-12345", "[0-9]{9}_prototype")

## Grade all directories (repositories) starting with '[0-9]*'
## as of 2020-04-30 23:59:59 using the configuration in file
## 'gradeconf-prototype'; write results in file 'GRADING.txt'
## of each directory.
grade("[0-9]*/", config_file = "gradeconf-prototype",
      time_limit = "2020-04-30 23:59:59",
      output_file = "GRADING.txt")

## Publish results in every repository in a new branch
## 'grading-results'. No need to specify the grading results file
## name since 'GRADING.txt' is the default in an English locale.
push("[0-9]*/", "grading-results", create_branch = TRUE,
     message = c("Here are your grading results"))
## End(Not run)
```

---

square_brackets_style *Validation of Spacing Around Square Brackets*

Description

Check that spacing around square brackets is valid.

Usage

```
open_bracket_style(srcData)
close_bracket_style(srcData)
```

Arguments

srcData	a list as returned by getSourceData.

Details

Good coding practices dictate the correct spacing around square brackets: opening brackets should not be immediately followed by a space; closing brackets should not be immediately preceded by a space, unless that space is after a comma.

Value

Boolean. When FALSE, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

lines	faulty line numbers;
message	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct use of spacing around square brackets
fil <- tempfile(fileext = ".R")
cat("x <- c(1, 2, 3, 5, 7, 9)",
    "x[x > 3 & x < 7]",
    "dim(x) <- c(2, 3)",
    "x[1, ]",
    file = fil, sep = "\n")
srcData <- getSourceData(fil)
open_bracket_style(srcData)
close_bracket_style(srcData)

## Incorrect use of spacing around square brackets
fil <- tempfile(fileext = ".R")
cat("x <- c(1, 2, 3, 5, 7, 9)",
    "x[ x > 3 & x < 7 ]",
    "dim(x) <- c(2, 3)",
    "x[1,]",
    "x[1,  ]",
    file = fil, sep = "\n")
srcData <- getSourceData(fil)
open_bracket_style(srcData)
close_bracket_style(srcData)
```

`trailing_blank_lines_style` *Validation of Trailing Blank Lines*

Description

Check that a script file does not contain superfluous trailing blank lines.

Usage

```
trailing_blank_lines_style(srcData)
```

Arguments

`srcData`	a list as returned by `getSourceData`.

Details

Good coding practices dictate that a script file should not end with blank lines.

Value

Boolean. When `FALSE`, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

`lines`	faulty line numbers;
`message`	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct script without trailing blank lines
fil <- tempfile(fileext = ".R")
cat("## A simple function",
    "foo <- function(x, y)",
    "{",
    "    x + y",
    "}",
    file = fil, sep = "\n")
trailing_blank_lines_style(getSourceData(fil))

## Incorrect script with trailing blank lines
fil <- tempfile(fileext = ".R")
cat("## A simple function",
    "foo <- function(x, y)",
    "{",
```

```
    "    x + y",
    "}",
    "",
    "",
    file = fil, sep = "\n")
trailing_blank_lines_style(getSourceData(fil))
```

`trailing_whitespace_style` *Validation of Trailing Whitespace*

Description

Check that a script file does not contain unnecessary whitespace at the end of lines.

Usage

```
trailing_whitespace_style(srcData)
```

Arguments

`srcData`	a list as returned by `getSourceData`.

Details

Good coding practices dictate that a script file should contain unnecessary whitespace (space or tabulation) at the end of lines.

Value

Boolean. When `FALSE`, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

`lines`	faulty line numbers;
`message`	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct script without trailing whitespace
fil <- tempfile(fileext = ".R")
cat("## A simple function",
    "foo <- function(x, y)",
    "{",
    "    x + y",
    "}",
```

```
    file = fil, sep = "\n")
trailing_whitespace_style(getSourceData(fil))

## Incorrect script with trailing whitespace
fil <- tempfile(fileext = ".R")
cat("## A simple function",
    "foo <- function(x, y)",
    "{   ",
    "    x + y",
    "}\t",
    file = fil, sep = "\n")
trailing_whitespace_style(getSourceData(fil))
```

`unneeded_concatenation_style` *Validation of Concatenation Usage*

Description

Check that function `c` is used with more than one argument.

Usage

```
unneeded_concatenation_style(srcData)
```

Arguments

`srcData`	a list as returned by `getSourceData`.

Details

Function `c` is used to combine its arguments. Therefore, good coding practice dictates that the function should never be used with zero or one argument.

Usage with zero argument to create an empty vector should be replaced by calls to object creation functions like `numeric` or `character`.

Usage with one argument is a superfluous call to `c` that should just be replaced by the argument.

Value

Boolean. When `FALSE`, a message indicates the nature of the error and the faulty lines, and the returned value has the following attributes:

`lines`	faulty line numbers;
`message`	text of the error message.

Examples

```
## Keep parse data in non interactive sessions.
if (!interactive())
    op <- options(keep.source = TRUE)

## Correct use of the 'c()' function
fil <- tempfile(fileext = ".R")
cat("x <- c(1, 2, 3, 4)", file = fil)
unneeded_concatenation_style(getSourceData(fil))

## Incorrect uses of the 'c()' function
fil <- tempfile(fileext = ".R")
cat("x <- c()",
    "x <- c(42)",
    file = fil, sep = "\n")
unneeded_concatenation_style(getSourceData(fil))
```

Index