Package: refactor (via r-universe)

March 5, 2026

Title Tools for Refactoring Code

Version 0.0.0.9000

Description Tools for refactoring code.

License GPL-3

Encoding UTF-8

Language en

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Imports waldo, pryr

URL https://github.com/moodymudskipper/refactor

BugReports https://github.com/moodymudskipper/refactor/issues

Config/pak/sysreqs libicu-dev

Repository https://cynkra.r-universe.dev

Date/Publication 2024-05-16 09:19:10 UTC

RemoteUrl https://github.com/moodymudskipper/refactor

RemoteRef HEAD

RemoteSha c788ba8a90386e85f4f4b4d2224172846e6afa82

Contents

check_files_parse 2
detect_similar_code 2
fetch_namespace_names 3
find_pkg_funs 3
identify_hybrid_scripts 4
refactor 4
use_lintr_template_on_file 6
use_namespace_check 6

Index **7**

check_files_parse *Check that files parse correctly*

Description

This identifies files that contain non syntactic code, including files that have an R extension despite not being an R script.

Usage

```
check_files_parse(path = ".", recursive = TRUE)
```

Arguments

path	A string. The path to a file or the folder to explore By default explores the working directory.
recursive	A boolean. Passed to `list.files()` if `path` is a directory

Value

Returns the path invisibly, called for side effects.

detect_similar_code *Detect similar code blocks*

Description

This is a wrapper around `dupree::dupree()`. It analyses provided files looking for similar code. It is a bit slow so it can be impractical on big projects to run it with default (all files contained in working directory recursively), in this case it is wiser to run it on one or more specific `paths` or/and or to filter the files using the `pattern` argument.

Usage

```
detect_similar_code(paths = ".", recursive = TRUE, pattern = NULL)
```

Arguments

paths	Paths to scripts or folders containing scripts
recursive	Whether folders should be explorer recursively
pattern	A regular expression used to filter files

Value

The `paths` argument invisibly. Called for side effects

fetch_namespace_names *Fetch namespace names*

Description

Scans the code and finds all namespaced call of the form `pkg::fun` and returns a vector of unique package names.

Usage

```
fetch_namespace_names(path = ".", recursive = TRUE)
```

Arguments

`path`	A string. The path to a file or the folder to explore By default explores the working directory.
`recursive`	A boolean. Passed to `list.files()` if `path` is a directory

Value

A character vector of package names

find_pkg_funs *Find all uses of a package's functions*

Description

This will show false positives, but guarantees that we don't miss any instance.

Usage

```
find_pkg_funs(pkg, path = ".", recursive = TRUE, exclude = NULL)
```

Arguments

`pkg`	A string. The name of the target package
`path`	A string. The path to a file or the folder to explore By default explores the working directory.
`recursive`	A boolean. Passed to `list.files()` if `path` is a directory
`exclude`	A character vector of function names to dismiss.

Value

Returns its input invisibly, called for side effects

identify_hybrid_scripts *Identify hybrid scripts*

Description

Identify scripts who contain both function definitions and other object definitions.

Usage

```
identify_hybrid_scripts(path = ".", recursive = TRUE)
```

Arguments

path	A string. The path to a file or the folder to explore By default explores the working directory.
recursive	A boolean. Passed to `list.files()` if `path` is a directory

Value

Returns the path invisibly, called for side effects.

refactor *Refactor Code*

Description

These operators are used to refactor code and differ in the difference of behavior they allow between refactored and original code.

Usage

```
original %refactor% refactored

original %refactor_chunk% refactored

original %refactor_value% refactored

original %refactor_chunk_and_value% refactored

original %refactor_chunk_efficiently% refactored

original %refactor_value_efficiently% refactored

original %refactor_chunk_and_value_efficiently% refactored
```

```
original %ignore_original% refactored

original %ignore_refactored% refactored
```

Arguments

original	original expression
refactored	refactored expression

Details

•

Both original and refactored expressions are run. By default the function will fail if the outputs are different. %ignore_original% and %ignore_refactored% do as heir names suggest.

Options can be set to alter the behavior of %refactor%:

- if refactor.value is TRUE (the default), the sameness of the outputs of original and refactored is tested
- if refactor.env is TRUE (default is FALSE), the sameness of the modifications to the local environment made by original and refactored is tested
- if refactor.time is TRUE (default is FALSE), the improved execution speed of the refactored solution is tested
- if refactor.waldo is TRUE (the default), the waldo::compare will be used to compare objects or environments in case of failure. 'waldo' is sometimes slow and if we set this option to FALSE, dplyr::all_equal() would be used instead.

%refactor_*% functions are variants that are not affected by options other than refactor.waldo:

- %refactor_chunk% behaves like %refactor% with options(refactor.value = FALSE, refactor.env = TRUE, refactor.time = FALSE), it's convenient to refactor chunks of code that modify the local environment.
- %refactor_value% behaves like %refactor% with options(refactor.value = TRUE, refactor.env = FALSE, refactor.time = FALSE), it's convenient to refactor the body of a function that returns a useful value.
- %refactor_chunk_and_value% behaves like %refactor% with options(refactor.value = TRUE, refactor.env = TRUE, refactor.time = FALSE), it's convenient to refactor the body of a function that returns a closure.
- %refactor_chunk_efficiently%, %refactor_value_efficiently% and %refactor_chunk_and_value_efficien are variants of the above which also check the improved execution speed of the refactored solution

2 additional functions are used to avoid akward commenting of code, when the original and refactored code have different behaviors.

- %ignore_original% and %ignore_refactored% are useful when original and refactored code give different results (possibly because one of them is wrong) and we want to keep both codes around without commenting.

use_lintr_template_on_file *Use lintr template*

Description

This opens up an untitled script in RStudio containing calls to `lintr::lint()` or `lintr::lint_dir()` with various linters, sorted by category and rough order of importance.

Usage

```
use_lintr_template_on_file(path = NULL)

use_lintr_template_on_dir(path = NULL)
```

Arguments

`path`	Path to a R script or a directory. By default `use_lint_template_on_file()` considers the active document and `use_lint_template_on_dir()` considers the project folder as returned by `here::here()`

Value

Returns `NULL` invisibly. Called for side effects.

use_namespace_check *Use namespace check*

Description

Wrapper around `fetch_namespace_names()` that opens a new RStudio source editot tab with code to be pasted at the top of the main script to enumerate required packages and test if they are installed.

Usage

```
use_namespace_check()
```

Value

Returns `NULL` invisibly, called for side effects.

Index

%ignore_original% *(*refactor*)*, 4
%ignore_refactored% *(*refactor*)*, 4
%refactor% *(*refactor*)*, 4
%refactor_chunk% *(*refactor*)*, 4
%refactor_chunk_and_value% *(*refactor*)*, 4
%refactor_chunk_and_value_efficiently%
        *(*refactor*)*, 4
%refactor_chunk_efficiently%
        *(*refactor*)*, 4
%refactor_value% *(*refactor*)*, 4
%refactor_value_efficiently%
        *(*refactor*)*, 4

check_files_parse, 2

detect_similar_code, 2

fetch_namespace_names, 3
find_pkg_funs, 3

identify_hybrid_scripts, 4

refactor, 4

use_lintr_template_on_dir
        *(*use_lintr_template_on_file*)*, 6
use_lintr_template_on_file, 6
use_namespace_check, 6