ropensci-review-tools

Release 0.0.1

Mark Padgham

Mar 26, 2024

OVERVIEW

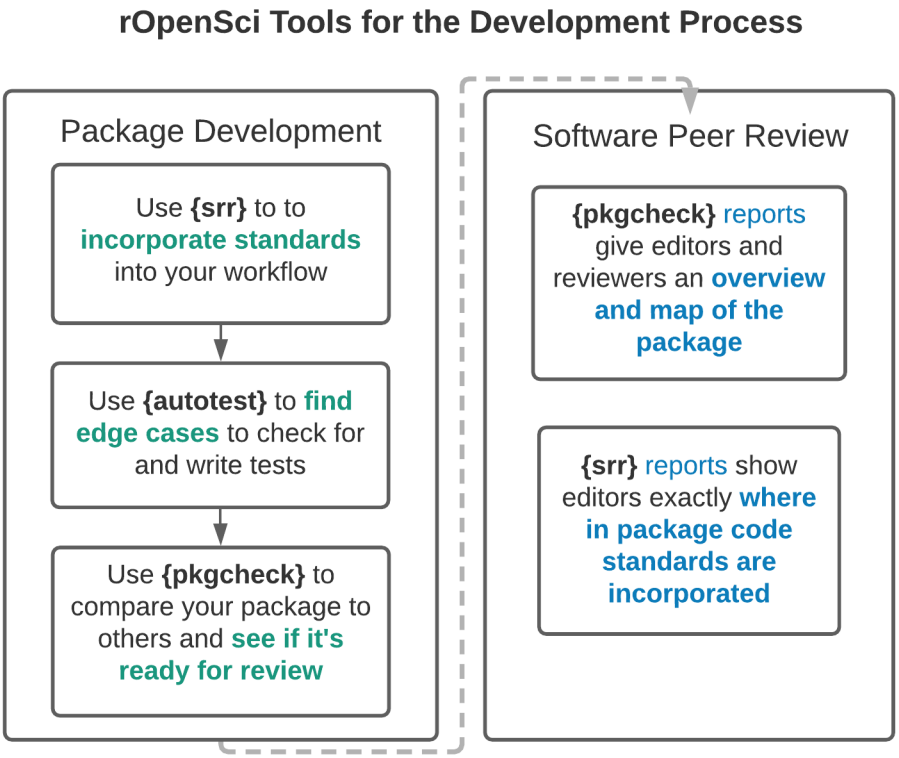
This organisation contains several packages developed for rOpenSci's review process, including packages specifically for the review of statistical software.

CHAPTER

ONE

OVERVIEW

This organisation contains several packages developed for rOpenSci's review process, and in particular for the review of statistical software. The following diagram depicts the relationships between some of the main packages:



Each package has its own repository, linked to below along with brief descriptions.

1.1 srr

The srr package helps authors document compliance with our standards for statistical software within their actual code. The package website has detailed descriptions of the procedure, including a demonstration which authors can first "walk through" to understand how the srr package works.

1.2 autotest

The autotest package is intended to be used from the first moments of package development, and throughout the preparation of packages for submission to our peer review system. It implements a form of mutation testing in an attempt to ensure all parameters of all functions respond appropriately to as many different forms and values of those parameters as possible. Continuous application of the autotest package throughout package development should ensure review processes are much less likely to uncover bugs in package behaviour.

1.3 pkgcheck

The pkgcheck package represents the final steps towards submitting a package for review with rOpenSci. Authors should only need the one function, pkgcheck(), which will confirm whether or not a package is ready to be submitted. The object returned by this function contains detailed information on various aspects of a package.

The following are links to the primary documentation for each package. Those wanting to use any one of these individual packages should follow the appropriate link.

CHAPTER

TWO

AUTOTEST

Automatic mutation testing of R packages. Mutation in the sense of mutating inputs (parameters) to function calls. `autotest` primarily works by scraping documented examples for all functions, and mutating the parameters input to those functions.

2.1 Installation

The easiest way to install this package is via the associated `r-universe`. As shown there, simply enable the universe with

```
options (repos = c (
    ropenscireviewtools = "https://ropensci-review-tools.r-universe.dev",
    CRAN = "https://cloud.r-project.org"
))
```

And then install the usual way with,

```
install.packages ("autotest")
```

Alternatively, the package can be installed by running one of the following lines:

```
# install.packages("remotes")
remotes::install_git ("https://git.sr.ht/~mpadge/autotest")
remotes::install_bitbucket ("mpadge/autotest")
remotes::install_gitlab ("mpadge/autotest")
remotes::install_github ("ropensci-review-tools/autotest")
```

The package can then be loaded the usual way:

```
library (autotest)
```

2.2 Usage

The simply way to use the package is

```
x <- autotest_package ("<package>")
```

The main argument to the `autotest_package()` function can either be the name of an installed package, or a path to a local directory containing the source for a package. The result is a `data.frame` of errors, warnings, and other diagnostic messages issued during package `auotest`-ing. The function has an additional parameter, `functions`, to restrict tests to specified functions only.

By default, `autotest_package()` returns a list of all tests applied to a package without actually running them. To implement those tests, set the parameter `test` to `TRUE`. Results are only returned for tests in which functions do not behave as expected, whether through triggering errors, warnings, or other behaviour as described below. The ideal behaviour of `autotest_package()` is to return nothing (or strictly, `NULL`), indicating that all tests passed successfully. See the main package vignette for an introductory tour of the package.

2.3 What is tested?

The package includes a function which lists all tests currently implemented.

```
autotest_types ()
#> # A tibble: 27 × 8
#>    type  test_name      fn_name parameter parameter_type operation content test
#>    <chr> <chr>          <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#>  1 dummy rect_as_other  <NA>    <NA>      rectangular    Convert ... "check... TRUE
#>  2 dummy rect_compare_... <NA>    <NA>      rectangular    Convert ... "expec... TRUE
#>  3 dummy rect_compare_... <NA>    <NA>      rectangular    Convert ... "expec... TRUE
#>  4 dummy rect_compare_... <NA>    <NA>      rectangular    Convert ... "expec... TRUE
#>  5 dummy extend_rect_c... <NA>    <NA>      rectangular    Extend e... "(Shou... TRUE
#>  6 dummy replace_rect_... <NA>    <NA>      rectangular    Replace ... "(Shou... TRUE
#>  7 dummy vector_to_lis... <NA>    <NA>      vector         Convert ... "(Shou... TRUE
#>  8 dummy vector_custom... <NA>    <NA>      vector         Custom c... "(Shou... TRUE
#>  9 dummy double_is_int  <NA>    <NA>      numeric        Check wh... "int p... TRUE
#> 10 dummy trivial_noise  <NA>    <NA>      numeric        Add triv... "(Shou... TRUE
#> #  17 more rows
```

That functions returns a `tibble` describing 27 unique tests. The default behaviour of `autotest_package()` with `test = FALSE` uses these test types to identify which tests will be applied to each parameter and function. The table returned from `autotest_types()` can be used to selectively switch tests off by setting values in the `test` column to `FALSE`, as demonstrated below.

2.4 How Does It Work?

The package works by scraping documented examples from all `.Rd` help files, and using those to identify the types of all parameters to all functions. Usage therefore first requires that the usage of all parameters be demonstrated in example code.

As described above, tests can also be selectively applied to particular functions through the parameters `functions`, used to nominate functions to include in tests, or `exclude`, used to nominate functions to exclude from tests. The following code illustrates.

```
x <- autotest_package (package = "stats", functions = "var", test = FALSE)
#>
#> ── autotesting stats ──
#>
#> ✓ [1 / 6]: var
#> ✓ [2 / 6]: cor
#> ✓ [3 / 6]: cor
#> ✓ [4 / 6]: cov
#> ✓ [5 / 6]: cov
#> ✓ [6 / 6]: cor
print (x)
#> # A tibble: 170 × 9
#>    type    test_name    fn_name parameter parameter_type operation content test
#>    <chr>   <chr>        <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#>  1 warning par_is_demo... var     use       <NA>           Check th... Exampl... TRUE
#>  2 warning par_is_demo... cov     y         <NA>           Check th... Exampl... TRUE
#>  3 dummy   trivial_noi... var     x         numeric        Add triv... (Shoul... TRUE
#>  4 dummy   vector_cust... var     x         vector         Custom c... (Shoul... TRUE
#>  5 dummy   vector_to_l... var     x         vector         Convert ... (Shoul... TRUE
#>  6 dummy   negate_logi... var     na.rm     single logical Negate d... (Funct... TRUE
#>  7 dummy   subst_int_f... var     na.rm     single logical Substitu... (Funct... TRUE
#>  8 dummy   subst_char_... var     na.rm     single logical Substitu... should... TRUE
#>  9 dummy   single_par_... var     na.rm     single logical Length 2... Should... TRUE
#> 10 dummy   return_succ... var     (return ... (return objec... Check th... <NA>   ␣
↪TRUE
#> #  160 more rows
#> #  1 more variable: yaml_hash <chr>
```

Testing the `var` function also tests `cor` and `cov`, because these are all documented within a single `.Rd` help file. Typing `?var` shows that the help topic is `cor`, and that the examples include the three functions, `var`, `cor`, and `cov`. That result details the 170 tests which would be applied to the `var` function from the `stats` package. These 170 tests yield the following results when actually applied:

```
y <- autotest_package (package = "stats", functions = "var", test = TRUE)
#> ── autotesting stats ──
#>
#> ✓ [1 / 6]: var
#> ✓ [2 / 6]: cor
#> ✓ [3 / 6]: cor
#> ✓ [4 / 6]: cov
#> ✓ [5 / 6]: cov
#> ✓ [6 / 6]: cor
print (y)
#> # A tibble: 25 × 9
#>    type       test_name fn_name parameter parameter_type operation content test
#>    <chr>      <chr>     <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#>  1 warning    par_is_d... var     use       <NA>           Check th... "Examp... TRUE
#>  2 warning    par_is_d... cov     y         <NA>           Check th... "Examp... TRUE
#>  3 diagnostic vector_t... var     x         vector         Convert ... "Funct... TRUE
#>  4 diagnostic subst_in... var     na.rm     single logical Substitu... "(Func... TRUE
#>  5 diagnostic vector_t... var     x         vector         Convert ... "Funct... TRUE
#>  6 diagnostic vector_t... var     y         vector         Convert ... "Funct... TRUE
#>  7 diagnostic single_c... cor     use       single charac... upper-ca... "is ca...␣
```

(continues on next page)

(continued from previous page)

```
↪TRUE
#>  8 diagnostic single_c... cor     method    single charac... upper-ca... "is ca...␣
↪TRUE
#>  9 diagnostic vector_c... cor     x         vector         Custom c... "Funct... TRUE
#> 10 diagnostic vector_c... cor     x         vector         Custom c... "Funct... TRUE
#> #  15 more rows
#> #  1 more variable: yaml_hash <chr>
```

And only 25 of the original 170 tests produced unexpected behaviour. There were in fact only 5 kinds of tests which produced these 25 results:

```
unique (y$operation)
#> [1] "Check that parameter usage is demonstrated"
#> [2] "Convert vector input to list-columns"
#> [3] "Substitute integer values for logical parameter"
#> [4] "upper-case character parameter"
#> [5] "Custom class definitions for vector input"
```

One of these involves conversion of a vector to a list-column representation (via `I(as.list(<vec>))`). Relatively few packages accept this kind of input, even though doing so is relatively straightforward. The following lines demonstrate how these tests can be switched off when `autotest`-ing a package. The `autotest_types()` function, used above to extract information on all types of tests, also accepts a single argument listing the `test_name` entries of any tests which are to be switched off.

```
types <- autotest_types (notest = "vector_to_list_col")
y <- autotest_package (
    package = "stats", functions = "var",
    test = TRUE, test_data = types
)
#> ── autotesting stats ──
#>
#> ✓ [1 / 6]: var
#> ✓ [2 / 6]: cor
#> ✓ [3 / 6]: cor
#> ✓ [4 / 6]: cov
#> ✓ [5 / 6]: cov
#> ✓ [6 / 6]: cor
print (y)
#> # A tibble: 22 × 9
#>    type       test_name fn_name parameter parameter_type operation content test
#>    <chr>      <chr>     <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#>  1 warning    par_is_d... var     use       <NA>           Check th... Exampl... TRUE
#>  2 warning    par_is_d... cov     y         <NA>           Check th... Exampl... TRUE
#>  3 diagnostic subst_in... var     na.rm     single logical Substitu... (Funct... TRUE
#>  4 diagnostic single_c... cor     use       single charac... upper-ca... is cas...␣
↪TRUE
#>  5 diagnostic single_c... cor     method    single charac... upper-ca... is cas...␣
↪TRUE
#>  6 diagnostic vector_c... cor     x         vector         Custom c... Functi... TRUE
#>  7 diagnostic vector_c... cor     x         vector         Custom c... Functi... TRUE
#>  8 diagnostic single_c... cor     method    single charac... upper-ca... is cas...␣
↪TRUE
```

(continues on next page)

(continued from previous page)

```
#>  9 diagnostic single_c... cor     use       single charac... upper-ca... is cas...␣
↪TRUE
#> 10 diagnostic single_c... cor     use       single charac... upper-ca... is cas...␣
↪TRUE
#> #  12 more rows
#> #  1 more variable: yaml_hash <chr>
```

Those tests are still returned from `autotest_package()`, but with `test = FALSE` to indicate they were not run, and a `type` of "no_test" rather than the previous "diagnostic".

2.5 Can `autotest` automatically create tests in my `tests` directory?

Not yet, but that should be possible soon. In the meantime, there are `testthat` expectations, listed in the main package functions, which enable `autotest` to be used in a package's test suite.

2.6 Prior work

1. The `great-expectations` framework for python, described in this medium article.
2. `QuickCheck` for Haskell
3. `mutate` for ruby
4. `mutant` for mutation of R code itself

2.7 Code of Conduct

Please note that this package is released with a Contributor Code of Conduct. By contributing to this project, you agree to abide by its terms.

2.8 Contributors

All contributions to this project are gratefully acknowledged using the `allcontributors` package following the all-contributors specification. Contributions of any kind are welcome!

2.8.1 Code

2.8.2 Issue Authors

2.8.3 Issue Contributors

2.9 Functions

2.9.1 at_yaml_template

at_yaml_template

Description

Generate a 'yaml' template for an 'autotest'.

Usage

```
at_yaml_template(loc = tempdir())
```

Arguments

Argument	Description
loc	Location to generate template file. Append with filename and '.yaml' suffix to overwrite default name of 'autotest.yaml', otherwise this parameter will be used to specify directory only.

Seealso

Other yaml: *autotest_yaml* , *examples_to_yaml*

2.9.2 autotest_obj

autotest_obj class definition

Description

This function exists only to provide the class definitions for test objects, and is not intended to be called directly.

Usage

```
autotest_obj(
  package = NA_character_,
  package_loc = NULL,
  test_name = NA_character_,
  fn_name = NA_character_,
  parameters = list(),
  parameter_types = NA_character_,
  class = NULL,
  classes = NULL,
  env = new.env(),
  test = FALSE,
  quiet = FALSE
)
```

Arguments

Argument	Description
package	Name of package for which object is to be constructed.
package_loc	Location of package on local system (for source packages only)
test_name	Name of test (use *autotest_types* to get all test names).
fn_name	Name of function to be tested.
parameters	Names of all parameters for that function.
parameter_type	Types of input parameters.
class	Class of an individual parameter.
classes	Classes of all parameters.
env	Environment in which tests are to be run.
test	If FALSE , return only descriptions of tests which would be run with test = TRUE , without actually running them.
quiet	If FALSE , issue progress and other messages during testing of object.

2.9.3 autotest_package

autotest_package

Description

Automatically test an entire package by converting examples to yaml format and submitting each to the *autotest_yaml* function.

Usage

```
autotest_package(
  package = ".",
  functions = NULL,
  exclude = NULL,
  test = FALSE,
  test_data = NULL,
  quiet = FALSE
)
```

Arguments

Argument	Description
package	Name of package, as either

- Path to local package source
- Name of installed package
- Full path to location of installed package if not on *.libPaths* , or
- Default which presumes current directory is within package to be tested. functions | Optional character vector containing names of functions of nominated package to be included in 'autotesting'. exclude | Optional character vector containing names of any functions of nominated package to be excluded from 'autotesting'. test | If FALSE , return only descriptions of tests which would be run with test = TRUE , without actually running them. test_data | Result returned from calling either *autotest_types* or *autotest_package* with test = FALSE that contains a list of all tests which would be conducted. These tests have an additional flag, test , which defaults to TRUE . Setting any tests to FALSE will avoid running them when test = TRUE . quiet | If 'FALSE', provide printed output on screen.

Value

An autotest_package object which is derived from a tibble tbl_df object. This has one row for each test, and the following nine columns:

- type The type of result, either "dummy" for test = FALSE , or one of "error", "warning", "diagnostic", or "message".
- test_name Name of each test
- fn_name Name of function being tested
- parameter Name of parameter being tested
- parameter_type Expected type of parameter as identified by autotest .
- operation Description of the test
- content For test = FALSE , the expected behaviour of the test; for test = TRUE , the observed discrepancy with that expected behaviour
- test If FALSE (default), list all tests without implementing them, otherwise implement all tests.
- yaml_hash' A unique hash which may be be used to extract the yaml` specification of each test. Some columns may contain NA values, as explained in the Note.

Seealso

Other main_functions: *autotest_types*

Note

Some columns may contain NA values, including:

- `parameer` and `parameter_type` , for tests applied to entire functions, such as tests of return values.
- `test_name` for warnings or errors generated through "normal" function calls generated directly from example code, in which case `type` will be "warning" or "error", and `content` will contain the content of the corresponding message.

2.9.4 autotest_types

autotest_types

Description

List all types of 'autotests' currently implemented.

Usage

```
autotest_types(notest = NULL)
```

Arguments

Argument	Description
`notes`	Character string of names of tests which should be switched off by setting the `test` column to `FALSE` . Run this function first without this parameter to get all names, then re-run with this parameter switch specified tests off.

Value

An `autotest` object with each row listing one unique type of test which can be applied to every parameter (of the appropriate class) of each function.

Seealso

Other main_functions: *autotest_package*

2.9.5 autotest_yaml

autotest_yaml

Description

Automatically test inputs to functions specified in a 'yaml' template.

Usage

```
autotest_yaml(
  yaml = NULL,
  filename = NULL,
  test = TRUE,
  test_data = NULL,
  quiet = FALSE
)
```

Arguments

Argument	Description
yaml	A 'yaml' template as a character vector, either hand-coded or potentially loaded via *readLines* function or similar. Should generally be left at default of 'NULL', with template specified by 'filename' parameter.
filen	Name (potentially including path) of file containing 'yaml' template. See *at_yaml_template* for details of template. Default uses template generated by that function, and held in local './tests' directory.
test	If FALSE , return only descriptions of tests which would be run with test = TRUE , without actually running them.
test_	Result returned from calling either *autotest_types* or *autotest_package* with test = FALSE that contains a list of all tests which would be conducted. These tests have an additional flag, test , which defaults to TRUE . Setting any tests to FALSE will avoid running them when test = TRUE .
quiet	If 'FALSE', provide printed output on screen.

Value

An autotest_pkg object, derived from a tibble , detailing instances of unexpected behaviour for every parameter of every function.

Seealso

Other yaml: *at_yaml_template* , *examples_to_yaml*

Examples

```
yaml_list <- examples_to_yaml (package = "stats", functions = "reshape")
res <- autotest_yaml (yaml = yaml_list)
```

2.9.6 autotest-package

autotest: Automatic Package Testing

Description

Automatic testing of R packages via a simple YAML schema.

Seealso

Useful links:

- https://docs.ropensci.org/autotest/
- https://github.com/ropensci-review-tools/autotest
- Report bugs at https://github.com/ropensci-review-tools/autotest/issues

Author

Maintainer : Mark Padgham mark.padgham@email.com

2.9.7 examples_to_yaml

examples_to_yaml

Description

Convert examples for a specified package, optionally restricted to one or more specified functions, to a list of 'autotest' 'yaml' objects to use to automatically test package.

Usage

```
examples_to_yaml(
  package = NULL,
  functions = NULL,
  exclude = NULL,
  quiet = FALSE
)
```

Arguments

Argument	Description
package	Name of package, as either

- Path to local package source
- Name of installed package
- Full path to location of installed package if not on *.libPaths* , or
- Default which presumes current directory is within package to be tested. `functions` | If specified, names of functions from which examples are to be obtained. `exclude` | Names of functions to exclude from 'yaml' template `quiet` | If 'FALSE', provide printed output on screen.

Seealso

Other yaml: *at_yaml_template* , *autotest_yaml*

2.9.8 expect_autotest_no_err

expect_autotest_no_err

Description

Expect `autotest_package()` to be clear of errors

Usage

```
expect_autotest_no_err(object)
```

Arguments

Argument	Description
object	An autotest object to be tested

Value

(invisibly) The same object

Seealso

Other expectations: *expect_autotest_no_testdata* , *expect_autotest_no_warn* , *expect_autotest_notes* , *expect_autotest_testdata*

2.9.9 expect_autotest_no_testdata

expect_autotest_no_testdata

Description

Expect autotest_package() to be clear of errors with no tests switched off

Usage

```
expect_autotest_no_testdata(object = NULL)
```

Arguments

Argument	Description
object	Not used here, but required for testthat expectations

Value

(invisibly) The autotest object

Seealso

Other expectations: *expect_autotest_no_err* , *expect_autotest_no_warn* , *expect_autotest_notes* , *expect_autotest_testdata*

2.9.10 expect_autotest_no_warn

expect_autotest_no_warn

Description

Expect `autotest_package()` to be clear of warnings

Usage

```
expect_autotest_no_warn(object)
```

Arguments

Argument	Description
`object`	An `autotest` object to be tested

Value

(invisibly) The same object

Seealso

Other expectations: *expect_autotest_no_err* , *expect_autotest_no_testdata* , *expect_autotest_notes* , *expect_autotest_testdata*

2.9.11 expect_autotest_notes

expect_autotest_notes

Description

Expect `test_data` param of `autotest_package` to have additional `note` column explaining why tests have been switched off.

Usage

```
expect_autotest_notes(object)
```

Arguments

Argument	Description
`object`	An `autotest` object to be tested

Seealso

Other expectations: *expect_autotest_no_err* , *expect_autotest_no_testdata* , *expect_autotest_no_warn* , *expect_autotest_testdata*

2.9.12 expect_autotest_testdata

expect_autotest_testdata

Description

Expect `autotest_package()` to be clear of errors with some tests switched off, and to have note column explaining why those tests are not run.

Usage

```
expect_autotest_testdata(object)
```

Arguments

Argument	Description
`object`	An `autotest_package` object with a `test` column flagging tests which are not to be run on the local package.

Value

(invisibly) The autotest object

Seealso

Other expectations: *expect_autotest_no_err* , *expect_autotest_no_testdata* , *expect_autotest_no_warn* , *expect_autotest_notes*

2.10 Vignettes

The first vignette demonstrates the process of applying `autotest` at all stages of package development. This vignette provides additional information for those applying `autotest` to already developed packages, in particular through describing how tests can be selectively applied to a package. By default the `autotest_package()` function tests an entire package, but testing can also be restricted to specified functions only. This vignette will demonstrate application to a few functions from the `stats` package, starting by loading the package.

```
library (autotest)
```

2.10.1 1. `.Rd` files, example code, and the autotest workflow

To understand what `autotest` does, it is first necessary to understand a bit about the structure of documentation files for R package, which are contained in files called `".Rd"` files. Tests are constructed by parsing individual `.Rd` documentation files to extract the example code, identifying parameters passed to the specified functions, and mutating those parameters.

The general procedure can be illustrated by examining a specific function, for which we now choose the `cor` function, because of its relative simplicity. The following lines extract the documentation for the `cor` function, a `.html` version of which can be seen by clicking on the link above. Note that that web page reveals the name of the `.Rd` file to be "cor" (in the upper left corner), meaning that the name of the `.Rd` file is `"cor.Rd"`. The following lines extract that content, first by loading the entire `.Rd` database for the `stats` package.

```
rd <- tools::Rd_db (package = "stats")
cor_rd <- rd [[grep ("^cor\\.Rd", names (rd))]]
```

The database itself is a list, with each entry holding the contents of one `.Rd` file in an object of class Rd, which is essentially a large nested list of components corresponding to the various `.Rd` tags such as `\arguments`, `\details`, and `\value`. An internal function from the `tools` package can be used to extract individual components (using the `:::` notation to access internal functions). For example, a single `.Rd` file often describes the functionality of several functions, each of which is identified by specifying the function name as an `"alias"`. The aliases for the `"cor.Rd"` file are:

```
tools:::.Rd_get_metadata (cor_rd, "alias")
#> [1] "var"     "cov"     "cor"     "cov2cor"
```

This one file thus contains documentation for those four functions. Example code can be extracted with a dedicated function from the `tools` package:

```
tools::Rd2ex (cor_rd)
```

```
#> ### Name: cor
#> ### Title: Correlation, Variance and Covariance (Matrices)
#> ### Aliases: var cov cor cov2cor
#> ### Keywords: univar multivariate array
#>
```

(continues on next page)

(continued from previous page)

```
#> ### ** Examples
#>
#> var(1:10)  # 9.166667
#>
#> var(1:5, 1:5) # 2.5
#>
#> ## Two simple vectors
#> cor(1:10, 2:11) # == 1
#>
#> ## Correlation Matrix of Multivariate sample:
#> (Cl <- cor(longley))
#> ## Graphical Correlation Matrix:
#> symnum(Cl) # highly correlated
#>
#> ## Spearman's rho  and  Kendall's tau
#> symnum(clS <- cor(longley, method = "spearman"))
#> symnum(clK <- cor(longley, method = "kendall"))
#> ## How much do they differ?
#> i <- lower.tri(Cl)
#> cor(cbind(P = Cl[i], S = clS[i], K = clK[i]))
#>
#>
#> ## cov2cor() scales a covariance matrix by its diagonal
#> ##           to become the correlation matrix.
#> cov2cor # see the function definition {and learn ..}
#> stopifnot(all.equal(Cl, cov2cor(cov(longley))),
#>           all.equal(cor(longley, method = "kendall"),
#>             cov2cor(cov(longley, method = "kendall"))))
#>
#> ##--- Missing value treatment:
#> C1 <- cov(swiss)
#> range(eigen(C1, only.values = TRUE)$values) # 6.19        1921
#>
#> ## swM := "swiss" with  3 "missing"s :
#> swM <- swiss
#> colnames(swM) <- abbreviate(colnames(swiss), minlength=6)
#> swM[1,2] <- swM[7,3] <- swM[25,5] <- NA # create 3 "missing"
#>
#> ## Consider all 5 "use" cases :
#> (C. <- cov(swM)) # use="everything"  quite a few NA's in cov.matrix
#> try(cov(swM, use = "all")) # Error: missing obs...
#> C2 <- cov(swM, use = "complete")
#> stopifnot(identical(C2, cov(swM, use = "na.or.complete")))
#> range(eigen(C2, only.values = TRUE)$values) # 6.46   1930
#> C3 <- cov(swM, use = "pairwise")
#> range(eigen(C3, only.values = TRUE)$values) # 6.19   1938
#>
#> ## Kendall's tau doesn't change much:
#> symnum(Rc <- cor(swM, method = "kendall", use = "complete"))
#> symnum(Rp <- cor(swM, method = "kendall", use = "pairwise"))
#> symnum(R. <- cor(swiss, method = "kendall"))
#>
```

(continues on next page)

(continued from previous page)

```
#> ## "pairwise" is closer componentwise,
#> summary(abs(c(1 - Rp/R.)))
#> summary(abs(c(1 - Rc/R.)))
#>
#> ## but "complete" is closer in Eigen space:
#> EV <- function(m) eigen(m, only.values=TRUE)$values
#> summary(abs(1 - EV(Rp)/EV(R.)) / abs(1 - EV(Rc)/EV(R.)))
```

This is the entire content of the `\examples` portion of `"cor.Rd"`, as can be confirmed by comparing with the online version.

2.10.2 2. Internal structure of the `autotest` workflow

For each `.Rd` file in a package, `autotest` tests the code given in the example section according to the following general steps:

1. Extract example lines from the `.Rd` file, as demonstrated above;
2. Identify all function `aliases` described by that file;
3. Identify all points at which those functions are called;
4. Identify all objects passed to those values, including values, classes, attributes, and other properties.
5. Identify any other parameters not explicitly passed in example code, but defined via default value;
6. Mutate the values of all parameters according to the kinds of test described in `autotest_types()`.

Calling `autotest_package(..., test = FALSE)` implements the first 5 of those 6 steps, and returns data on all possible mutations of each parameter, while setting `test = TRUE` actually passes the mutated parameters to the specified functions, and returns reports on any unexpected behaviour.

2.10.3 3. `autotest`-ing the `stats::cov` function

The preceding sections describe how `autotest` actually works, while the present section demonstrates how the package is typically used in practice. As demonstrated in the `README`, information on all tests implemented within the package can be obtained by calling the `autotest_types()` function. The main function for testing package is `autotest_package()`. The nominated package can be either an installed package, or the full path to a local directory containing a package's source code. By default all `.Rd` files of a package are tested, with restriction to specified functions possible either by nominating functions to exclude from testing (via the `exclude` parameter), or functions to include (via the `functions` parameter). The `functions` parameter is intended to enable testing only of specified functions, while the `exclude` parameter is intended to enable testing of all functions except those specified with this parameter. Specifying values for both of these parameters is not generally recommended.

3.1 Listing tests without conducting them

The following demonstrates the results of `autotest`-ing the `cor` function of the `stats` package, noting that the default call uses `test = FALSE`, and so returns details of all tests without actually implementing them (and for this reason we name the object `xf` for "false"):

```
xf <- autotest_package (package = "stats", functions = "cor")
print (xf)
```

```
#> # A tibble: 15 × 8
#>    type  test_name      fn_name parameter parameter_type operation content test
#>    <chr> <chr>          <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#>  1 dummy single_char_c... cor     use       single_charac... lower-ca... (Shoul...␣
↪TRUE
#>  2 dummy single_char_c... cor     use       single_charac... upper-ca... (Shoul...␣
↪TRUE
#>  3 dummy single_par_as... cor     use       single_charac... Length 2... Should...␣
↪TRUE
#>  4 dummy return_succes... cor     (return ... (return objec... Check th... <NA>   ␣
↪TRUE
#>  5 dummy return_val_de... cor     (return ... (return objec... Check th... <NA>   ␣
↪TRUE
#>  6 dummy return_desc_i... cor     (return ... (return objec... Check wh... <NA>   ␣
↪TRUE
#>  7 dummy return_class_... cor     (return ... (return objec... Compare ... <NA>   ␣
↪TRUE
#>  8 dummy par_is_docume... cor     x         <NA>           Check th... <NA>    TRUE
#>  9 dummy par_matches_d... cor     x         <NA>           Check th... <NA>    TRUE
#> 10 dummy par_is_docume... cor     use       <NA>           Check th... <NA>    TRUE
#> 11 dummy par_matches_d... cor     use       <NA>           Check th... <NA>    TRUE
#> 12 dummy par_is_docume... cor     method    <NA>           Check th... <NA>    TRUE
#> 13 dummy par_matches_d... cor     method    <NA>           Check th... <NA>    TRUE
#> 14 dummy par_is_docume... cor     y         <NA>           Check th... <NA>    TRUE
#> 15 dummy par_matches_d... cor     y         <NA>           Check th... <NA>    TRUE
```

The object returned from `autotest_package()` is a simple `tibble`, with each row detailing one test which would be applied to the each of the listed functions and parameters. Because no tests were conducted, all tests will generally have a `type` of `"dummy"`. In this case, however, we see the following:

```
table (xf$type)
#>
#> dummy
#>    15
```

In addition to the 15 dummy tests, the function also returns 0 warnings, the corresponding rows of which are:

```
xf [xf$type != "dummy", c ("fn_name", "parameter", "operation", "content")]
#> # A tibble: 0 × 4
#> #  4 variables: fn_name <chr>, parameter <chr>, operation <chr>, content <chr>
```

Although the `auotest` package is primarily intended to apply mutation tests to all parameters of all functions of a package, doing so requires identifying parameter types and classes through parsing example code. Any parameters of a function which are neither demonstrated within example code, nor given default values can not be tested, because it is not possible to determine their expected types. The above result reveals that neither the `use` parameter of the `var`

function, nor the `y` parameter of `cov`, are demonstrated in example code, triggering a warning that these parameter are unable to be tested.

3.2 Conducting tests

The 15 tests listed above with `type == "dummy"` can then be applied to all nominated functions and parameters by calling the same function with `test = TRUE`. Doing so yields the following results (as an object names `xt` for "true"):

```
xt <- autotest_package (package = "stats",
                        functions = "cor",
                        test = TRUE)
print (xt)
```

And the 15 tests yielded 5 unexpected responses. The best way to understand these results is to examine the object in detail, typically through `edit(xt)`, or equivalently in RStudio, clicking on the listed object. The different types of tests which produced unexpected responses were:

```
table (xt1$operation)
#>
#> Check that documentation matches class of input parameter
#>                                                         4
#>                            upper-case character parameter
#>                                                         1
```

Two of those reflect the previous results regarding parameters unable to be tested, while the remainder come from only two types of tests. Information on the precise results is contained in the `content` column, although in this case it is fairly straightforward to see that the operation "upper case character parameter" arises because the `use` and `method` parameters of the `cor` and `cov` functions are case-dependent, and are only accepted in lower case form. The other operation is the conversion of vectors to list-column format, as described in the first vignette.

3.4 Controlling which tests are conducted

The `test` parameter of the `autotest_package()` function can be used to control whether all tests are conducted or not. Finer-level control over tests can be achieved by specifying the `test_data` parameter. This parameter must be an object of class `autotest_package`, as returned by either the `autotest_types()`] or `autotest_package()` functions. The former of these is the function which specifies all unique tests, and so returns a relatively small `tibble` of 27 rows. The following lines demonstrate how to switch off the list-column test for all functions and parameters:

```
types <- autotest_types()
types$test [grep ("list_col", types$test_name)] <- FALSE
xt2 <- autotest_package (package = "stats",
                         functions = "cor",
                         test = TRUE,
                         test_data = types)
print (xt2)
```

```
#> # A tibble: 5 × 8
#>   type       test_name  fn_name parameter parameter_type operation content test
#>   <chr>      <chr>      <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#> 1 warning    par_match... cor     x         <NA>           Check th... Parame... TRUE
#> 2 warning    par_match... cor     y         <NA>           Check th... Parame... TRUE
#> 3 warning    par_match... cor     x         <NA>           Check th... Parame... TRUE
```

(continues on next page)

(continued from previous page)

```
#> 4 warning    par_match... cor     y         <NA>           Check th... Parame... TRUE
#> 5 diagnostic single_ch... cor     use       single charac... upper-ca... is cas...␣
↪TRUE
```

The result now has four rows with `test == FALSE`, and `type == "no_test"`, indicating that these tests were not actually conducted. That also makes apparent the role of these `test` flags. When initially calling `autotest_package()` with default `test = FALSE`, the result contains a `test` column in which all values are `TRUE`. Although potentially perplexing at first, this value must be understood in relation to the `type` column. A `type` of `"dummy"` indicates that a test has not been conducted, in which case `test = TRUE` is a control flag used to determine what would be conducted if these data were submitted as the `test_data` parameter. For all `type` values other than `"dummy"`, the `test` column specifies whether or not each test was actually conducted.

The preceding example showed how the results of `autotest_types()` can be used to control which tests are implemented for an entire package. Finer-scale control can be achieved by modifying individual rows of the full table returned by `autotest_package()`. The following code demonstrates by showing how list-column tests can be switched off only for particular functions, starting again with the `xf` data of dummy tests generated above.

```
xf <- autotest_package (package = "stats",
                        functions = "cor")
xf$test [grepl ("list_col", xf$test_name) & xf$fn_name == "var"] <- FALSE
xt3 <- autotest_package (package = "stats",
                         functions = "cor",
                         test = TRUE,
                         test_data = xf)
print (xt3)
```

```
#> # A tibble: 5 × 8
#>   type       test_name  fn_name parameter parameter_type operation content test
#>   <chr>      <chr>      <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#> 1 warning    par_match... cor     x         <NA>           Check th... Parame... TRUE
#> 2 warning    par_match... cor     y         <NA>           Check th... Parame... TRUE
#> 3 warning    par_match... cor     x         <NA>           Check th... Parame... TRUE
#> 4 warning    par_match... cor     y         <NA>           Check th... Parame... TRUE
#> 5 diagnostic single_ch... cor     use       single charac... upper-ca... is cas...␣
↪TRUE
```

These procedures illustrate the three successively finer levels of control over tests, by switching them off for:

1. Entire packages;
2. Specified functions only; or
3. Specific parameters of particular functions only.

2.10.4 4. autotest-ing your package

`autotest` can be very easily incorporated in your package's `tests/` directory via to simple `testthat` expectations:

- `expect_autotest_no_testdata`, which will expect `autotest_package` to work on your package with default values including no additional `test_data` specifying tests which are not to be run; or
- `expect_autotest_testdata`, to be used when specific tests are switched off.

Using these requires adding `autotest` to the `Suggests` list of your package's `DESCRIPTION` file, along with `testthat`. Note that the use of testing frameworks other than `testthat` is possible through writing custom expectations for the

output of autotest_package(), but that is not considered here.

To use these expectations, you must first decide which, if any, tests you judge to be not applicable to your package, and switch them off following the procedure described above (that is, at the package level through modifying the test flag of the object returned from autotest_types(), or at finer function- or parameter-levels by modifying equivalent values in the object returned from autotest_package(..., test = FALSE). These objects must then be passed as the test_data parameter to autotest_package(). If you consider all tests to be applicable, then autotest_package() can be called without specifying this parameter.

If you switch tests off via a test_data parameter, then the expect_autotest expectation requires you to append an additional column to the test_data object called "note" (case-insensitive), and include a note for each row which has test = FALSE explaining why those tests have been switched off. Lines in your test directory should look something like this:

```
library (testthat) # as called in your test suite
# For example, to switch off vector-to-list-column tests:
test_data <- autotest_types (notest = "vector_to_list_col")
test_data$note <- ""
test_data$note [test_data$test == "vector_to_list_col"] <-
    "These tests are not applicable because ..."
expect_success (expect_autotest_testdata (test_data))
```

This procedure of requiring an additional "note" column ensures that your own test suite will explicitly include all explanations of why you deem particular tests not to be applicable to your package.

In contrast, the following expectation should be used when autotest_package() passes with all tests are implemented, in which case no parameters need be passed to the expectation, and tests will confirm that no warnings or errors are generated.

```
expect_success (expect_autotest_no_testdata ())
```

4.2 Finer control over testing expectations

The two expectations shown above call the autotest_package() function internally, and assert that the results follow the expected pattern. There are also three additional testthat expectations which can be applied to pre-generated autotest objects, to allow for finer control over testing expectations. These are:

- expect_autotest_no_err to expect no errors in results from autotest_package();
- expect_autotest_no_warn to expect no warnings; and
- expect_autotest_notes to expect tests which have been switched off to have an additional "note" column explaining why.

These tests are demonstrated in one of the testing files used in this package, which the following lines recreate to demonstrate the general process. The first two expectations are that an object be free from both warnings and errors. The tests implemented here are applied to the stats::cov() function, which actually triggers warnings because two parameters do not have their usage demonstrated in the example code. The tests therefore expect_failure(), when they generally should expect_success() throughout.

```
library (testthat) # as called in your test suite
# For example, to switch off vector-to-list-column tests:
test_data <- autotest_types (notest = "vector_to_list_col")
x <- autotest_package (package = "stats",
                       functions = "cov",
                       test = TRUE,
```

(continues on next page)

(continued from previous page)

```
                        test_data = test_data)

expect_success (expect_autotest_no_err (x))
expect_failure (expect_autotest_no_warn (x)) # should expect_success!!
```

The test files then affirms that simply passing the object, `x`, which has tests flagged as `type == "no_test"`, yet without explaining why in an additional `"note"` column, should cause `expect_autotest()` to fail. The following line, removing the logical `testthat` expectation, demonstrates:

```
expect_autotest_notes (x)
```

As demonstrated above, these `expect_autotest_...` calls should always be wrapped in a direct `testhat` expectation of `expect_success()`. To achieve success in that case, we need to append an additional `"note"` column containing explanations of why each test has been switched off:

```
x$note <- ""
x [grep ("vector_to_list", x$test_name), "note"] <-
  "these tests have been switched off because ..."

expect_success (expect_autotest_notes (x))
```

In general, using `autotest` in a package's test suite should be as simple as adding `autotest` to `Suggests`, and wrapping either `expect_autotest_no_testdata` or `expect_autotest_testdata` in an `expect_success` call.

2.10.5 How to use autotest

This vignette demonstrates the easiest way to use `autotest`, which is to apply it continuously through the entire process of package development. The best way to understand the process is to obtain a local copy of the vignette itself from this link, and step through the code. We begin by constructing a simple package in the local `tempdir()`.

To create a package in one simple line, we use `usethis::create_package()`, and name our package `"demo"`.

```
path <- file.path (tempdir (), "demo")
usethis::create_package (path, check_name = FALSE, open = FALSE)
#> ✓ Creating '/tmp/RtmpddpQhn/demo/'
#> ✓ Setting active project to '/tmp/RtmpddpQhn/demo'
#> ✓ Creating 'R/'
#> ✓ Writing 'DESCRIPTION'
#> Package: demo
#> Title: What the Package Does (One Line, Title Case)
#> Version: 0.0.0.9000
#> Authors@R (parsed):
#>     * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)
#> Description: What the package does (one paragraph).
#> License: `use_mit_license()`, `use_gpl3_license()` or friends to pick a
#>     license
#> Encoding: UTF-8
#> Roxygen: list(markdown = TRUE)
#> RoxygenNote: 7.2.3
#> ✓ Writing 'NAMESPACE'
#> ✓ Setting active project to '<no active project>'
```

The structure looks like this:

```
fs::dir_tree (path)
#> /tmp/RtmpddpQhn/demo
#> ├── DESCRIPTION
#> ├── NAMESPACE
#> └── R
```

Having constructed a minimal package structure, we can then insert some code in the R/ directory, including initial roxygen2 documentation lines, and use the roxygenise() function to create the corresponding man files.

autotest works by parsing and running "example" code from function documentation, so our code needs to include at least one example line.

```
code <- c ("#' my_function",
           "#'",
           "#' @param x An input",
           "#' @return Something else",
           "#' @examples",
           "#' y <- my_function (x = 1)",
           "#' @export",
           "my_function <- function (x) {",
           "  return (x + 1)",
           "}")
writeLines (code, file.path (path, "R", "myfn.R"))
roxygen2::roxygenise (path)
#>  Loading demo
#> Writing 'NAMESPACE'
#> Writing 'my_function.Rd'
```

Our package now looks like this:

```
fs::dir_tree (path)
#> /tmp/RtmpddpQhn/demo
#> ├── DESCRIPTION
#> ├── NAMESPACE
#> ├── R
#> │   └── myfn.R
#> └── man
#>     └── my_function.Rd
```

We can already apply autotest to that package to see what happens, first ensuring that we've loaded the package ready to use.

```
library (autotest)
x0 <- autotest_package (path)
```

```
#>  Loading autotest
#> ✓ [1 / 1]
```

We use the DT package to display the results here.

```
DT::datatable (x0, options = list (dom = "t")) # display table only
```

The first thing to notice is the first column, which has test_type = "dummy" for all rows. The autotest_package() function has a parameter test with a default value of FALSE, so that the default call demonstrated above does not

actually implement the tests, rather it returns an object listing all tests that would be performed with actually doing so. Applying the tests by setting `test = TRUE` gives the following result.

```
x1 <- autotest_package (path, test = TRUE)
#>  Loading demo
#> ✓ [1 / 1]
DT::datatable (x1, options = list (dom = "t"))
```

Of the 9 tests which were performed, only 3 yielded unexpected behaviour. The first indicates that the parameter `x` has only been used as an integer, yet was not specified as such. The second states that the parameter `x` is "assumed to be a single numeric". `autotest` does its best to figure out what types of inputs are expected for each parameter, and with the example only demonstrating `x = 1`, assumes that `x` is always expected to be a single value. We can resolve the first of these by replacing `x = 1` with `x = 1.` to clearly indicate that it is not an integer, and the second by asserting that `length(x) == 1`, as follows:

```
code <- c ("#' my_function",
           "#'",
           "#' @param x An input",
           "#' @return Something else",
           "#' @examples",
           "#' y <- my_function (x = 1.)",
           "#' @export",
           "my_function <- function (x) {",
           "  if (length(x) > 1) {",
           "    warning(\"only the first value of x will be used\")",
           "    x <- x [1]",
           "  }",
           "  return (x + 1)",
           "}")
writeLines (code, file.path (path, "R", "myfn.R"))
roxygen2::roxygenise (path)
#>  Loading demo
#> Writing 'my_function.Rd'
```

This is then sufficient to pass all `autotest` tests and so return `NULL`.

```
autotest_package (path, test = TRUE)
#> ✓ [1 / 1]
#> # A tibble: 3 × 8
#>   type       test_name  fn_name parameter parameter_type operation content test
#>   <chr>      <chr>      <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#> 1 error      <NA>       my_fun... <NA>      <NA>           normal f... ":quot... TRUE
#> 2 error      return_su... my_fun... (return ... (return objec... error fr...
↪"could... TRUE
#> 3 diagnostic return_de... my_fun... (return ... (return objec... Check wh...
↪"Funct... TRUE
```

Integer input

Note that `autotest` distinguishes integer and non-integer types by their `storage.mode` of `"integer"` and `"double"`, and not by their respective classes of `"integer"` and `"numeric"`, because `"numeric"` is ambiguous in R, and `is.numeric(1L)` is `TRUE`, even though `storage.mode(1L)` is `"integer"`, and not `"numeric"`. Replacing `x = 1` with `x = 1.` explicitly identifies that parameter as a `"double"` parameter, and allowed the preceding tests to pass. Note what happens if we instead specify that parameter as an integer (`x = 1L`).

```
code [6] <- gsub ("1\\.", "1L", code [6])
writeLines (code, file.path (path, "R", "myfn.R"))
roxygen2::roxygenise (path)
#>  Loading demo
#> Writing 'my_function.Rd'
x2 <- autotest_package (path, test = TRUE)
#> ✓ [1 / 1]
DT::datatable (x2, options = list (dom = "t"))
```

That then generates two additional messages, the second of which reflects an expectation that parameters assumed to be integer-valued should assert that, for example by converting with `as.integer()`. The following suffices to remove that message.

```
code <- c (code [1:12],
           "  if (is.numeric (x))",
           "    x <- as.integer (x)",
           code [13:length (code)])
```

The remaining message concerns integer ranges. For any parameters which `autotest` identifies as single integers, routines will try a full range of values between `+/- .Machine$integer.max`, to ensure that all values are appropriately handled. Many routines may sensibly allow unrestricted ranges, while many others may not implement explicit control over permissible ranges, yet may error on, for example, unexpectedly large positive or negative values. The content of the diagnostic message indicates one way to resolve this issue, which is simply by describing the input as `"unrestricted"`.

```
code [3] <- gsub ("An input", "An unrestricted input", code [3])
writeLines (code, file.path (path, "R", "myfn.R"))
roxygen2::roxygenise (path)
#>  Loading demo
#> Writing 'my_function.Rd'
autotest_package (path, test = TRUE)
#> ✓ [1 / 1]
#> # A tibble: 5 × 8
#>   type       test_name  fn_name parameter parameter_type operation content test
#>   <chr>      <chr>      <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#> 1 error      <NA>       my_fun... <NA>      <NA>           normal f... ":quot... TRUE
#> 2 error      <NA>       my_fun... <NA>      <NA>           <NA>      ":quot... TRUE
#> 3 error      return_su... my_fun... (return ... (return objec... error fr...
↪"could... TRUE
#> 4 diagnostic int_range  my_fun... x         single integer Ascertai... "Funct... TRUE
#> 5 diagnostic return_de... my_fun... (return ... (return objec... Check wh...
↪"Funct... TRUE
```

An alternative, and frequently better way, is to ensure and document specific control over permissible ranges, as in the following revision of our function.

```
code <- c ("#' my_function",
           "#'",
           "#' @param x An input between 0 and 10",
           "#' @return Something else",
           "#' @examples",
           "#' y <- my_function (x = 1L)",
           "#' @export",
           "my_function <- function (x) {",
           "  if (length(x) > 1) {",
           "    warning(\"only the first value of x will be used\")",
           "    x <- x [1]",
           "  }",
           "  if (is.numeric (x))",
           "    x <- as.integer (x)",
           "  if (x < 0 | x > 10) {",
           "    stop (\"x must be between 0 and 10\")",
           "  }",
           "  return (x + 1L)",
           "}")
writeLines (code, file.path (path, "R", "myfn.R"))
roxygen2::roxygenise (path)
#>  Loading demo
#> Writing 'my_function.Rd'
autotest_package (path, test = TRUE)
#> ✓ [1 / 1]
#> # A tibble: 5 × 8
#>   type       test_name  fn_name parameter parameter_type operation content test
#>   <chr>      <chr>      <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#> 1 error      <NA>       my_fun... <NA>      <NA>           normal f... ":quot... TRUE
#> 2 error      <NA>       my_fun... <NA>      <NA>           <NA>      ":quot... TRUE
#> 3 error      return_su... my_fun... (return ... (return objec... error fr...
↪"could... TRUE
#> 4 diagnostic int_range  my_fun... x         single integer Ascertai... "Funct... TRUE
#> 5 diagnostic return_de... my_fun... (return ... (return objec... Check wh...
↪"Funct... TRUE
```

Respective limits of ranges may be specified with any of the following words:

- Lower limits: “more”, “greater”, “larger than”, “lower limit of”, “above”
- Upper limits: “less”, “lower”, “smaller than”, “upper limit of”, “below”

Vector input

The initial test results above suggested that the input was *assumed* to be of length one. Let us now revert our function to its original format which accepted vectors of length > 1, and include an example demonstrating such input.

```
code <- c ("#' my_function",
           "#'",
           "#' @param x An input",
           "#' @return Something else",
           "#' @examples",
           "#' y <- my_function (x = 1)",
```

(continues on next page)

(continued from previous page)

```
          "#' y <- my_function (x = 1:2)",
          "#' @export",
          "my_function <- function (x) {",
          "  if (is.numeric (x)) {",
          "    x <- as.integer (x)",
          "  }",
          "  return (x + 1L)",
          "}")
writeLines (code, file.path (path, "R", "myfn.R"))
roxygen2::roxygenise (path)
#>  Loading demo
#> Writing 'my_function.Rd'
```

Note that the first example no longer has `x = 1L`. This is because vector inputs are identified as `integer` by examining all individual values, and presuming `integer` representations for any parameters for which all values are whole numbers, regardless of `storage.mode`.

```
x3 <- autotest_package (path, test = TRUE)
#> ✓ [1 / 2]
#> ✓ [2 / 2]
DT::datatable (x3, options = list (dom = "t"))
```

List-column conversion

The above result reflects one of the standard tests, which is to determine whether list-column formats are appropriately processed. List-columns commonly arise when using (either directly or indirectly), the `tidyr::nest()` function, or equivalently in base R with the `I` or `AsIs` function. They look like this:

```
dat <- data.frame (x = 1:3, y = 4:6)
dat$x <- I (as.list (dat$x)) # base R
dat <- tidyr::nest (dat, y = y)
print (dat)
#> # A tibble: 3 × 2
#>   x         y
#>   <I<list>> <list>
#> 1 <int [1]> <tibble [1 × 1]>
#> 2 <int [1]> <tibble [1 × 1]>
#> 3 <int [1]> <tibble [1 × 1]>
```

The use of packages like `tidyr` and `purrr` quite often leads to `tibble`-class inputs which contain list-columns. Any functions which fail to identify and appropriately respond to such inputs may generate unexpected errors, and this `autotest` is intended to enforce appropriate handling of these kinds of inputs. The following lines demonstrate the kinds of results that can arise without such checks.

```
m <- mtcars
head (m, n = 2L)
#>               mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
m$mpg <- I (as.list (m$mpg))
head (m, n = 2L) # looks exaxtly the same
```

(continues on next page)

(continued from previous page)

```
#>               mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
cor (m)
#> Error in cor(m): 'x' must be numeric
```

In contrast, many functions either assume inputs to be lists, and convert when not, or implicitly `unlist`. Either way, such functions may respond entirely consistently regardless of the presence of list-columns, like this:

```
m$mpg <- paste0 ("a", m$mpg)
class (m$mpg)
#> [1] "character"
```

The list-column `autotest` is intended to enforce consistent behaviour in response to list-column inputs. One way to identify list-column formats is to check the value of `class(unclass(.))` of each column. The `unclass` function is necessary to first remove any additional class attributes, such as `I` in `dat$x` above. A modified version of our function which identifies and responds to list-column inputs might look like this:

```
code <- c ("#' my_function",
           "#'",
           "#' @param x An input",
           "#' @return Something else",
           "#' @examples",
           "#' y <- my_function (x = 1)",
           "#' y <- my_function (x = 1:2)",
           "#' @export",
           "my_function <- function (x) {",
           "  if (methods::is (unclass (x), \"list\")) {",
           "    x <- unlist (x)",
           "  }",
           "  if (is.numeric (x)) {",
           "    x <- as.integer (x)",
           "  }",
           "  return (x + 1L)",
           "}")
writeLines (code, file.path (path, "R", "myfn.R"))
roxygen2::roxygenise (path)
#>  Loading demo
```

That change once again leads to clean `autotest` results:

```
autotest_package (path, test = TRUE)
#> ✓ [1 / 2]
#> ✓ [2 / 2]
#> # A tibble: 5 × 8
#>   type       test_name  fn_name parameter parameter_type operation content test
#>   <chr>      <chr>      <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#> 1 error      <NA>       my_fun... <NA>      <NA>           <NA>      ":quot... TRUE
#> 2 error      <NA>       my_fun... <NA>      <NA>           normal f... ":quot... TRUE
#> 3 error      return_su... my_fun... (return ... (return objec... error fr...
↪"could... TRUE
#> 4 error      <NA>       my_fun... <NA>      <NA>           normal f... ":quot... TRUE
#> 5 diagnostic return_de... my_fun... (return ... (return objec... Check wh...
```

(continues on next page)

(continued from previous page)

```
↪"Funct... TRUE
```

Of course simply attempting to `unlist` a complex list-column may be dangerous, and it may be preferable to issue some kind of message or warning, or even either simply remove any list-columns entirely or generate an error. Replacing the above, potentially dangerous, line, `x <- unlist (x)` with a simple `stop("list-columns are not allowed")` will also produce clean `autotest` results.

Return results and documentation

Functions which return complicated results, such as objects with specific classes, need to document those class types, and `autotest` compares return objects with documentation to ensure that this is done. The following code constructs a new function to demonstrate some of the ways `autotest` inspects return objects, demonstrating a vector input (`length(x) > 1`) in the example to avoid messages regarding length checks an integer ranges.

```
code <- c ("#' my_function3",
           "#'",
           "#' @param x An input",
           "#' @examples",
           "#' y <- my_function3 (x = 1:2)",
           "#' @export",
           "my_function3 <- function (x) {",
           "  return (datasets::iris)",
           "}")
writeLines (code, file.path (path, "R", "myfn3.R"))
roxygen2::roxygenise (path) # need to update docs with seed param
#>  Loading demo
#> Writing 'NAMESPACE'
#> Writing 'my_function3.Rd'
x4 <- autotest_package (path, test = TRUE)
#> ✓ [1 / 2]
#> ✓ [2 / 2]
DT::datatable (x4, options = list (dom = "t"))
```

Several new diagnostic messages are then issued regarding the description of the returned value. Let's insert a description to see the effect.

```
code <- c (code [1:3],
           "#' @return The iris data set as dataframe",
           code [4:length (code)])
writeLines (code, file.path (path, "R", "myfn3.R"))
roxygen2::roxygenise (path) # need to update docs with seed param
#>  Loading demo
#> Writing 'my_function3.Rd'
x5 <- autotest_package (path, test = TRUE)
#> ✓ [1 / 2]
#> ✓ [2 / 2]
DT::datatable (x5, options = list (dom = "t"))
```

That result still contains a couple of diagnostic messages, but it is now pretty clear what we need to do, which is to be precise with our specification of the class of return object. The following then suffices to once again generate clean `autotest` results.

```
code [4] <- "#' @return The iris data set as data.frame"
writeLines (code, file.path (path, "R", "myfn3.R"))
roxygen2::roxygenise (path) # need to update docs with seed param
#>  Loading demo
#> Writing 'my_function3.Rd'
autotest_package (path, test = TRUE)
#> ✓ [1 / 2]
#> ✓ [2 / 2]
#> # A tibble: 5 × 8
#>   type       test_name  fn_name parameter parameter_type operation content test
#>   <chr>      <chr>      <chr>   <chr>     <chr>          <chr>     <chr>   <lgl>
#> 1 error      <NA>       my_fun... <NA>      <NA>           normal f... ":quot... TRUE
#> 2 error      return_su... my_fun... (return ... (return objec... error fr...
↪"could... TRUE
#> 3 error      <NA>       my_fun... <NA>      <NA>           <NA>      ":quot... TRUE
#> 4 error      <NA>       my_fun... <NA>      <NA>           normal f... ":quot... TRUE
#> 5 diagnostic return_de... my_fun... (return ... (return objec... Check wh...
↪"Funct... TRUE
```

Documentation of input parameters

Similar checks are performed on the documentation of input parameters, as demonstrated by the following modified version of the preceding function.

```
code <- c ("#' my_function3",
           "#'",
           "#' @param x An input",
           "#' @return The iris data set as data.frame",
           "#' @examples",
           "#' y <- my_function3 (x = datasets::iris)",
           "#' @export",
           "my_function3 <- function (x) {",
           "  return (x)",
           "}")
writeLines (code, file.path (path, "R", "myfn3.R"))
roxygen2::roxygenise (path) # need to update docs with seed param
#>  Loading demo
#> Writing 'my_function3.Rd'
x6 <- autotest_package (path, test = TRUE)
#> ✓ [1 / 2]
#> ✓ [2 / 2]
DT::datatable (x6, options = list (dom = "t"))
```

This warning again indicates precisely how it can be rectified, for example by replacing the third line with

```
code [3] <- "#' @param x An input which can be a data.frame"
```

General Procedure

The demonstrations above hopefully suffice to indicate the general procedure which `autotest` attempts to make as simple as possible. This procedure consists of the following single point:

- From the moment you develop your first function, and every single time you modify your code, do whatever steps are necessary to ensure `autotest_package()` returns `NULL`.

This vignette has only demonstrated a few of the tests included in the package, but as long as you use `autotest` throughout the entire process of package development, any additional diagnostic messages should include sufficient information for you to be able to restructure your code to avoid them.

CHAPTER

THREE

PKGSTATS

Extract summary statistics of R package structure and functionality. Not all statistics of course, but a good go at balancing insightful statistics while ensuring computational feasibility. `pkgstats` is a *static* code analysis tool, so is generally very fast (a few seconds at most for very large packages). Installation is described in a separate vignette.

3.1 What statistics?

Statistics are derived from these primary sources:

1. Numbers of lines of code, documentation, and white space (both between and within lines) in each directory and language
2. Summaries of package `DESCRIPTION` file and related package meta-statistics
3. Summaries of all objects created via package code across multiple languages and all directories containing source code (`./R`, `./src`, and `./inst/include`).
4. A function call network derived from function definitions obtained from the code tagging library, `ctags`, and references ("calls") to those obtained from another tagging library, `gtags`. This network roughly connects every object making a call (as `from`) with every object being called (`to`).
5. An additional function call network connecting calls within R functions to all functions from other R packages.

The primary function, `pkgstats()`, returns a list of these various components, including full `data.frame` objects for the final three components described above. The statistical properties of this list can be aggregated by the `pkgstats_summary()` function, which returns a `data.frame` with a single row of summary statistics. This function is demonstrated below, including full details of all statistics extracted.

3.2 Demonstration

The following code demonstrates the output of the main function, `pkgstats`, using an internally bundled `.tar.gz` "tarball" of this package. The `system.time` call demonstrates that the static code analyses of `pkgstats` are generally very fast.

```
library (pkgstats)
tarball <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
system.time (
    p <- pkgstats (tarball)
)
```

```
##    user  system elapsed
##   1.701   0.124   1.802
```

```
names (p)
```

```
## [1] "loc"           "vignettes"      "data_stats"     "desc"
## [5] "translations"  "objects"        "network"        "external_calls"
```

The result is a list of various data extracted from the code. All except for `objects` and `network` represent summary data:

```
p [!names (p) %in% c ("objects", "network", "external_calls")]
```

```
## $loc
## # A tibble: 3 × 12
## # Groups:   language, dir [3]
##   language dir   nfiles nlines ncode  ndoc nempty nspaces nchars nexpr ntabs
##   <chr>    <chr>  <int>  <int> <int> <int>  <int>   <int>  <int> <dbl> <int>
## 1 C++      src        3    365   277    21     67     933   7002     1     0
## 2 R        R         19   3741  2698   536    507   27575  94022     1     0
## 3 R        tests      7    348   266    10     72     770   6161     1     0
## # ... with 1 more variable: indentation <int>
##
## $vignettes
## vignettes     demos
##         0         0
##
## $data_stats
##           n  total_size median_size
##           0           0           0
##
## $desc
##    package version                date license
## 1 pkgstats     9.9 2022-05-12 11:41:22   GPL-3
##                                                                                 ↵
↪urls
## 1 https://docs.ropensci.org/pkgstats/,\nhttps://github.com/ropensci-review-tools/
↪pkgstats
##                                                      bugs aut ctb fnd rev ths
## 1 https://github.com/ropensci-review-tools/pkgstats/issues   1   0   0   0   0
##   trl depends                                                        imports
## 1   0      NA brio, checkmate, dplyr, fs, igraph, methods, readr, sys, withr
##                                                                      suggests
## 1 hms, knitr, pbapply, pkgbuild, Rcpp, rmarkdown, roxygen2, testthat, visNetwork
##   enhances linking_to
## 1       NA      cpp11
##
## $translations
## [1] NA
```

The various components of these results are described in further detail in the main package vignette.

3.2.1 Overview of statistics and the `pkgstats_summary()` function

A summary of the `pkgstats` data can be obtained by submitting the object returned from `pkgstats()` to the `pkgstats_summary()` function:

```
s <- pkgstats_summary (p)
```

This function reduces the result of the `pkgstats()` function to a single line with 95 entries, represented as a `data.frame` with one row and that number of columns. This format is intended to enable summary statistics from multiple packages to be aggregated by simply binding rows together. While 95 statistics might seem like a lot, the `pkgstats_summary()` function aims to return as many usable raw statistics as possible in order to flexibly allow higher-level statistics to be derived through combination and aggregation. These 95 statistics can be roughly grouped into the following categories (not shown in the order in which they actually appear), with variable names in parentheses after each description. Some statistics are summarised as comma-delimited character strings, such as translations into human languages, or other packages listed under "depends", "imports", or "suggests". This enables subsequent analyses of their contents, for example of actual translated languages, or both aggregate numbers and individual details of all package dependencies, as demonstrated immediately below.

Package Summaries

- name (`package`)
- Package version (`version`)
- Package date, as modification time of `DESCRIPTION` file where not explicitly stated (`date`)
- License (`license`)
- Languages, as a single comma-separated character value (`languages`), and excluding `R` itself.
- List of translations where package includes translations files, given as list of (spoken) language codes (`translations`).

Information from `DESCRIPTION` file

- Package URL(s) (`url`)
- URL for BugReports (`bugs`)
- Number of contributors with role of *author* (`desc_n_aut`), *contributor* (`desc_n_ctb`), *funder* (`desc_n_fnd`), *reviewer* (`desc_n_rev`), *thesis advisor* (`ths`), and *translator* (`trl`, relating to translation between computer and not spoken languages).
- Comma-separated character entries for all `depends`, `imports`, `suggests`, and `linking_to` packages.

Numbers of entries in each the of the last two kinds of items can be obtained from by a simple `strsplit` call, like this:

```
deps <- strsplit (s$suggests, ", ") [[1]]
length (deps)
```

```
## [1] 9
```

```
print (deps)
```

```
## [1] "hms"        "knitr"      "pbapply"    "pkgbuild"   "Rcpp"
## [6] "rmarkdown"  "roxygen2"   "testthat"   "visNetwork"
```

Numbers of files and associated data

- Number of vignettes (`num_vignettes`)

- Number of demos (num_demos)
- Number of data files (num_data_files)
- Total size of all package data (data_size_total)
- Median size of package data files (data_size_median)
- Numbers of files in main sub-directories (files_R, files_src, files_inst, files_vignettes, files_tests), where numbers are recursively counted in all sub-directories, and where inst only counts files in the inst/include sub-directory.

Statistics on lines of code

- Total lines of code in each sub-directory (loc_R, loc_src, loc_ins, loc_vignettes, loc_tests).
- Total numbers of blank lines in each sub-directory (blank_lines_R, blank_lines_src, blank_lines_inst, blank_lines_vignette, blank_lines_tests).
- Total numbers of comment lines in each sub-directory (comment_lines_R, comment_lines_src, comment_lines_inst, comment_lines_vignettes, comment_lines_tests).
- Measures of relative white space in each sub-directory (rel_space_R, rel_space_src, rel_space_inst, rel_space_vignettes, rel_space_tests), as well as an overall measure for the R/, src/, and inst/ directories (rel_space).
- The number of spaces used to indent code (indentation), with values of -1 indicating indentation with tab characters.
- The median number of nested expression per line of code, counting only those lines which have any expressions (nexpr).

Statistics on individual objects (including functions)

These statistics all refer to "functions", but actually represent more general "objects," such as global variables or class definitions (generally from languages other than R), as detailed below.

- Numbers of functions in R (n_fns_r)
- Numbers of exported and non-exported R functions (n_fns_r_exported, n_fns_r_not_exported)
- Number of functions (or objects) in other computer languages (n_fns_src), including functions in both src and inst/include directories.
- Number of functions (or objects) per individual file in R and in all other (src) directories (n_fns_per_file_r, n_fns_per_file_src).
- Median and mean numbers of parameters per exported R function (npars_exported_mn, npars_exported_md).
- Mean and median lines of code per function in R and other languages, including distinction between exported and non-exported R functions (loc_per_fn_r_mn, loc_per_fn_r_md, loc_per_fn_r_exp_m, loc_per_fn_r_exp_md, loc_per_fn_r_not_exp_mn, loc_per_fn_r_not_exp_m, loc_per_fn_src_mn, loc_per_fn_src_md).
- Equivalent mean and median numbers of documentation lines per function (doclines_per_fn_exp_mn, doclines_per_fn_exp_md, doclines_per_fn_not_exp_m, doclines_per_fn_not_exp_md, docchars_per_par_exp_mn, docchars_per_par_exp_m).

Network Statistics

The full structure of the network table is described below, with summary statistics including:

- Number of edges, including distinction between languages (n_edges, n_edges_r, n_edges_src).
- Number of distinct clusters in package network (n_clusters).

- Mean and median centrality of all network edges, calculated from both directed and undirected representations of network (`centrality_dir_mn`, `centrality_dir_md`, `centrality_undir_mn`, `centrality_undir_md`).
- Equivalent centrality values excluding edges with centrality of zero (`centrality_dir_mn_no0`, `centrality_dir_md_no0`, `centrality_undir_mn_no0`, `centrality_undir_md_no`).
- Numbers of terminal edges (`num_terminal_edges_dir`, `num_terminal_edges_undir`).
- Summary statistics on node degree (`node_degree_mn`, `node_degree_md`, `node_degree_max`)

External Call Statistics

The final column in the result of the `pkgstats_summary()` function summarises the `external_calls` object detailing all calls make to external packages (including to base and recommended packages). This summary is also represented as a single character string. Each package lists total numbers of function calls, and total numbers of unique function calls. Data for each package are separated by a comma, while data within each package are separated by a colon.

```
s$external_calls
```

```
## [1] "base:447:78,brio:7:1,dplyr:7:4,fs:4:2,graphics:10:2,hms:1:1,igraph:3:3,
↪pbapply:1:1,pkgstats:99:60,readr:8:5,stats:16:2,sys:13:1,tools:2:2,utils:10:7,
↪visNetwork:3:2,withr:5:1"
```

This structure allows numbers of calls to all packages to be readily extracted with code like the following:

```
calls <- do.call (
    rbind,
    strsplit (strsplit (s$external_call, ",") [[1]], ":")
)
calls <- data.frame (
    package = calls [, 1],
    n_total = as.integer (calls [, 2]),
    n_unique = as.integer (calls [, 3])
)
print (calls)
```

```
##       package n_total n_unique
## 1        base     447       78
## 2        brio       7        1
## 3       dplyr       7        4
## 4          fs       4        2
## 5    graphics      10        2
## 6         hms       1        1
## 7      igraph       3        3
## 8     pbapply       1        1
## 9    pkgstats      99       60
## 10      readr       8        5
## 11      stats      16        2
## 12        sys      13        1
## 13      tools       2        2
## 14      utils      10        7
## 15 visNetwork       3        2
## 16      withr       5        1
```

The two numeric columns respectively show the total number of calls made to each package, and the total number of

unique functions used within those packages. These results provide detailed information on numbers of calls made to, and functions used from, other R packages, including base and recommended packages.

Finally, the summary statistics conclude with two further statistics of `afferent_pkg` and `efferent_pkg`. These are package-internal measures of afferent and efferent couplings between the files of a package. The *afferent* couplings (`ca`) are numbers of *incoming* calls to each file of a package from functions defined elsewhere in the package, while the *efferent* couplings (`ce`) are numbers of *outgoing* calls from each file of a package to functions defined elsewhere in the package. These can be used to derive a measure of "internal package instability" as the ratio of efferent to total coupling (`ce / (ce + ca)`).

There are many other "raw" statistics returned by the main `pkgstats()` function which are not represented in `pkgstats_summary()`. The main package vignette provides further detail on the full results.

The following sub-sections provide further detail on the `objects`, `network`, and `external_call` items, which could be used to extract additional statistics beyond those described here.

3.3 Code of Conduct

Please note that this package is released with a Contributor Code of Conduct. By contributing to this project, you agree to abide by its terms.

3.4 Functions

3.4.1 `ctags_install`

Install 'ctags' from a clone of the 'git' repository

Description

'ctags' is installed with this package on both Windows and macOS systems; this is an additional function to install from source on Unix systems.

Usage

```
ctags_install(bin_dir = NULL, sudo = TRUE)
```

Arguments

Argument	Description
`bin_di`	Prefix to pass to the `autoconf` configure command defining location to install the binary, with default of /usr/local .
`sudo`	Set to `FALSE` if `sudo` is not available, in which case a value for `bin_dir` will also have to be explicitly specified, and be a location where a binary is able to be installed without `sudo` privileges.

Value

Nothing; the function will fail if installation fails, otherwise returns nothing.

Seealso

Other tags: *ctags_test* , *tags_data*

Examples

```
ctags_install (bin_dir = "/usr/local") # default
```

3.4.2 ctags_test

test a 'ctags' installation

Description

This uses the example from https://github.com/universal-ctags/ctags/blob/master/man/ctags-lang-r.7.rst.in and also checks the GNU global installation.

Usage

```
ctags_test(quiet = TRUE)
```

Arguments

Argument	Description
quiet	If TRUE , display on screen whether or not 'ctags' is correctly installed.

Value

'TRUE' or 'FALSE' respectively indicating whether or not 'ctags' is correctly installed.

Seealso

Other tags: *ctags_install* , *tags_data*

Examples

```
ctags_test ()
```

3.4.3 desc_stats

Statistics from DESCRIPTION files

Description

Statistics from DESCRIPTION files

Usage

```
desc_stats(path)
```

Arguments

Argument	Description
path	Directory to source code of package being analysed

Value

A data.frame with one row and 16 columns extracting various information from the 'DESCRIPTION' file, include websites, tallies of different kinds of authors and contributors, and package dependencies.

Seealso

Other stats: *loc_stats* , *pkgstats_summary* , *pkgstats* , *rd_stats*

Examples

```
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
# have to extract tarball to call function on source code:
path <- extract_tarball (f)
desc_stats (path)
```

3.4.4 dl_pkgstats_data

Download latest version of 'pkgstats' data

Description

Download latest version of 'pkgstats' data

Usage

```
dl_pkgstats_data(current = TRUE, path = tempdir(), quiet = FALSE)
```

Arguments

Argument	Description
current	If 'FALSE', download data for all CRAN packages ever released, otherwise (default) download data only for current CRAN packages.
path	Local path to download file.
quiet	If FALSE , display progress information on screen.

Value

(Invisibly) A data.frame of pkgstats results, one row for each package.

Seealso

Other archive: *pkgstats_fns_from_archive* , *pkgstats_from_archive*

3.4.5 extract_tarball

Extract tarball of a package into temp directory and return path to extracted package

Description

Extract tarball of a package into temp directory and return path to extracted package

Usage

```
extract_tarball(tarball)
```

Arguments

Argument	Description
tarball	Full path to local tarball of an R package.

Value

Path to extracted version of package (in `tempdir()`).

Seealso

Other misc: *pkgstats_fn_names*

Examples

```
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
path <- extract_tarball (f)
```

3.4.6 loc_stats

Internal calculation of Lines-of-Code Statistics

Description

Internal calculation of Lines-of-Code Statistics

Usage

```
loc_stats(path)
```

Arguments

Argument	Description
path	Directory to source code of package being analysed

Value

A list of statistics for each of three directories, 'R', 'src', and 'inst/include', each one having 5 statistics of total numbers of lines, numbers of empty lines, total numbers of white spaces, total numbers of characters, and indentation used in files in that directory.

Seealso

Other stats: *desc_stats* , *pkgstats_summary* , *pkgstats* , *rd_stats*

Note

NA values are returned for directories which do not exist.

Examples

```
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
# have to extract tarball to call function on source code:
path <- extract_tarball (f)
loc_stats (path)
```

3.4.7 pkgstats_fn_names

Extract names of all functions for one R package

Description

Extract names of all functions for one R package

Usage

```
pkgstats_fn_names(path)
```

Arguments

Argument	Description
path	Either a path to a local source repository, or a local '.tar.gz' file, containing code for an R package.

Value

A `data.frame` with three columns:

- package: Name of package
- version: Package version
- fn_name: Name of function

Seealso

Other misc: *extract_tarball*

Examples

```
# 'path' can be path to a package tarball:
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
path <- extract_tarball (f)
s <- pkgstats_fn_names (path)
```

3.4.8 pkgstats_fns_from_archive

Trawl a local CRAN archive to extract function names only from all packages

Description

Trawl a local CRAN archive to extract function names only from all packages

Usage

```
pkgstats_fns_from_archive(
  path,
  archive = FALSE,
  prev_results = NULL,
  results_file = NULL,
  chunk_size = 1000L,
  num_cores = 1L,
  results_path = tempdir()
)
```

Arguments

Argument	Description
path	Path to local archive of R packages, either as source directories, or '.tar.gz' files such as in a CRAN mirror.
archi	If TRUE , extract statistics for all packages in the /Archive sub-directory, otherwise only statistics for main directory (that is, current packages only).
prev_	Result of previous call to this function, if available. Submitting previous results will ensure that only newer packages not present in previous result will be analysed, with new results simply appended to previous results. This parameter can also specify a file to be read with readRDS() .
resul	Can be used to specify the name or full path of a .Rds file to which results should be saved once they have been generated. The '.Rds' extension will be automatically appended, and any other extensions will be ignored.
chunk	Divide large archive trawl into chunks of this size, and save intermediate results to local files. These intermediate files can be combined to generate a single prev_results file, to enable jobs to be stopped and re-started without having to recalculate all results. These files will be named pkgstats-results-N.Rds , where "N" incrementally numbers each file.
num_c	Number of machine cores to use in parallel, defaulting to single-core processing.
resul	Path to save intermediate files generated by the chunk_size parameter described above.

Value

A data.frame object with one row for each function in each package and the following columns:

- Package name
- Package version
- Function name

Seealso

Other archive: *dl_pkgstats_data* , *pkgstats_from_archive*

3.4.9 pkgstats_from_archive

Trawl a local CRAN archive and extract statistics from all packages

Description

Trawl a local CRAN archive and extract statistics from all packages

Usage

```
pkgstats_from_archive(
  path,
  archive = TRUE,
  prev_results = NULL,
  results_file = NULL,
  chunk_size = 1000L,
  num_cores = 1L,
  save_full = FALSE,
  save_ex_calls = FALSE,
  results_path = tempdir()
)
```

Arguments

Argument	Description
path	Path to local archive of R packages, either as source directories, or '.tar.gz' files such as in a CRAN mirror.
archi	If TRUE , extract statistics for all packages in the /Archive sub-directory, otherwise only statistics for main directory (that is, current packages only).
prev_	Result of previous call to this function, if available. Submitting previous results will ensure that only newer packages not present in previous result will be analysed, with new results simply appended to previous results. This parameter can also specify a file to be read with readRDS() .
resul	Can be used to specify the name or full path of a .Rds file to which results should be saved once they have been generated. The '.Rds' extension will be automatically appended, and any other extensions will be ignored.
chunk	Divide large archive trawl into chunks of this size, and save intermediate results to local files. These intermediate files can be combined to generate a single prev_results file, to enable jobs to be stopped and re-started without having to recalculate all results. These files will be named pkgstats-results-N.Rds , where "N" incrementally numbers each file.
num_c	Number of machine cores to use in parallel, defaulting to single-core processing.
save_	If TRUE , full *pkgstats* results are saved for each package to files in results_path .
save_	If TRUE , the results of the external_calls component are saved for each package to files in results_path (only if save_full = FALSE).
resul	Path to save intermediate files generated by the chunk_size parameter described above.

Value

A `data.frame` object with one row for each package containing summary statistics generated from the *pkgstats_summary* function.

Seealso

Other archive: *dl_pkgstats_data* , *pkgstats_fns_from_archive*

Examples

```
# Create fake archive directory with single tarball:
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
tarball <- basename (f)

archive_path <- file.path (tempdir (), "archive")
if (!dir.exists (archive_path)) {
dir.create (archive_path)
}
path <- file.path (archive_path, tarball)
file.copy (f, path)
tarball_path <- file.path (archive_path, "tarballs")
dir.create (tarball_path, recursive = TRUE)
file.copy (path, file.path (tarball_path, tarball))
out <- pkgstats_from_archive (tarball_path)
```

3.4.10 pkgstats_summary

Condense the output of `pkgstats` to summary statistics only

Description

Condense the output of `pkgstats` to summary statistics only

Usage

```
pkgstats_summary(s = NULL)
```

Arguments

Argument	Description
s	Output of *pkgstats* , containing full statistical data on one package. Default of `NULL` returns a single row with `NA` values (used in *pkgstats_from_archive*).

Value

Summarised version of `s` , as a single row of a standardised `data.frame` object

Seealso

Other stats: *desc_stats* , *loc_stats* , *pkgstats* , *rd_stats*

Note

Variable names in the summary object use the following abbreviations:

- "loc" = Lines-of-Code
- "fn" = Function
- "n_fns" = Number of functions
- "npars" = Number of parameters
- "doclines" = Number of documentation lines
- "nedges" = Number of edges in function call network, as a count of unique edges, which may be less than the size of the `network` object returned by *pkgstats* , because that may include multiple calls between identical function pairs.
- "n_clusters" = Number of connected clusters within the function call network.
- "centrality" used as a prefix for several statistics, along with "dir" or "undir" for centrality calculated on networks respectively constructed with directed or undirected edges; "mn" or "md" for respective measures of mean or median centrality, and "no0" for measures excluding edges with zero centrality.

Examples

```
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
p <- pkgstats (f)
s <- pkgstats_summary (p)
```

3.4.11 pkgstats-package

pkgstats: Metrics of R Packages

Description

Static code analyses for R packages using the external code-tagging libraries "ctags" and "gtags". Static analyses enable packages to be analysed very quickly, generally a couple of seconds at most. The package also provides access to a database generating by applying the main function to the full CRAN archive, enabling the statistical properties of any package to be compared with all other CRAN packages.

Seealso

Useful links:

- https://docs.ropensci.org/pkgstats/
- https://github.com/ropensci-review-tools/pkgstats
- Report bugs at https://github.com/ropensci-review-tools/pkgstats/issues

Author

Maintainer : Mark Padgham mark.padgham@email.com (ORCID)

3.4.12 pkgstats

Analyse statistics of one R package

Description

Analyse statistics of one R package

Usage

```
pkgstats(path = ".")
```

Arguments

Argument	Description
path	Either a path to a local source repository, or a local '.tar.gz' file, containing code for an R package.

Value

List of statistics and data on function call networks (or object relationships in other languages). Includes the following components:

- list("loc: ") list("Summary of Lines-of-Code in all package directories")
- list("vignettes: ") list("Numbers of vignettes and "demo" files")
- list("data_stats: ") list("Statistics of numbers and sizes of package data files")
- list("desc: ") list("Summary of contents of 'DESCRIPTION' file")
- list("translations: ") list("List of translations into other (human) languages\n", "(where provides)")
- list("objects: ") list("A ", list("data.frame"), " of all functions in R, and all other\n", "objects (functions, classes, structures, global variables, and more) in all\n", "other languages")
- list("network: ") list("A ", list("data.frame"), " of object references within and between all\n", "languages; in R these are function calls, but may be more abstract in other\n", "languages.")

- list("external_calls: ") list("A ", list("data.frame"), " of all calls make to all functions\n", "from all other R packages, including base and recommended as well as\n", "contributed packages.")

Seealso

Other stats: *desc_stats* , *loc_stats* , *pkgstats_summary* , *rd_stats*

Examples

```
# 'path' can be path to a package tarball:
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
s <- pkgstats (f)
# or to a source directory:
path <- extract_tarball (f)
s <- pkgstats (path)
```

3.4.13 plot_network

Plot interactive visNetwork visualisation of object-relationship network of package.

Description

Plot interactive visNetwork visualisation of object-relationship network of package.

Usage

```
plot_network(s, plot = TRUE, vis_save = NULL)
```

Arguments

Argument	Description
s	Package statistics obtained from *pkgstats* function.
plot	If TRUE , plot the network using visNetwork which opens an interactive browser pane.
vis_save	Name of local file in which to save html file of network visualisation (will override plot to FALSE).

Value

(Invisibly) A visNetwork representation of the package network.

Note

Edge thicknesses are scaled to centrality within the package function call network. Node sizes are scaled to numbers of times each function is called from all other functions within a package.

Examples

```
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
p <- pkgstats (f)
plot_network (p)
```

3.4.14 rd_stats

Stats from '.Rd' files

Description

Stats from '.Rd' files

Usage

```
rd_stats(path)
```

Arguments

Argument	Description
path	Directory to source code of package being analysed

Value

A data.frame of function names and numbers of parameters and lines of documentation for each, along with mean and median numbers of characters used to document each parameter.

Seealso

Other stats: *desc_stats* , *loc_stats* , *pkgstats_summary* , *pkgstats*

Examples

```
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
# have to extract tarball to call function on source code:
path <- extract_tarball (f)
rd_stats (path)
```

3.4.15 tags_data

use ctags and gtags to parse call data

Description

use ctags and gtags to parse call data

Usage

```
tags_data(path, has_tabs = NULL, pkg_name = NULL)
```

Arguments

Argument	Description
path	Path to local repository
has_t	A logical flag indicating whether or not the code contains any tab characters. This can be determined from *loc_stats* , which has a `tabs` column. If not given, that value will be extracted from internally calling that function.
pkg_n	Only used for external_call_network, to label package-internal calls.

Value

A list of three items:

- "network" A `data.frame` of relationships between objects, generally as calls between functions in R, but other kinds of relationships in other source languages. This is effectively an edge-based network representation, and the data frame also include network metrics for each edge, calculated through representing the network in both directed (suffix "_dir") and undirected (suffix "_undir") forms.
- "objects" A `data.frame` of statistics on each object (generally functions in R, and other kinds of objects in other source languages), including the kind of object, the language, numbers of lines-of-code, parameters, and lines of documentation, and a binary flag indicating whether or not R functions accept "three-dots" parameters (`...`).
- "external_calls" A `data.frame` of every call from within every R function to any external R package, including base and recommended packages. The location of each calls is recorded, along with the external function and package being called.

Seealso

Other tags: *ctags_install* , *ctags_test*

Examples

```
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
# have to extract tarball to call function on source code:
path <- extract_tarball (f)
tags <- tags_data (path)
```

3.5 Vignettes

3.5.1 Installation

The `pkgstats` package is on CRAN, so can be installed directly with,

```
install.packages ("pkgstats")
```

The latest development version can be installed via the associated `r-universe`. As shown there, simply enable the universe with

```
options (repos = c (
    ropenscireviewtools = "https://ropensci-review-tools.r-universe.dev",
    CRAN = "https://cloud.r-project.org"
))
```

And then call `install.packages()` the same way. Alternatively, the development version of the package can be installed by running one of the following lines:

```
remotes::install_github ("ropensci-review-tools/pkgstats")
pak::pkg_install ("ropensci-review-tools/pkgstats")
```

The package can then loaded for use with:

```
library (pkgstats)
```

Installation on Linux systems

This package requires the system libraries `ctags-universal` and GNU `global`, both of which are automatically installed along with the package on both Windows and MacOS systems. Most Linux distributions do not include a sufficiently up-to-date version of `ctags-universal`, and so it must be compiled from source. This can be done by running a single function, `ctags_install()`, which will install both `ctags-universal` and GNU `global`.

The `pkgstats` package includes a function to ensure your local installations of `universal-ctags` and `global` work correctly. Please ensure you see the following prior to proceeding:

```
ctags_test ()
#> [1] TRUE
```

Note that GNU `global` can be linked at installation to the Universal Ctags plug-in parser to expand the default 5 languages to 30. This makes no difference to `pkgstats` results, as `gtags` output is only used to trace function call networks, which is only possible for compiled languages able to dynamically share pointers to the same objects. This is possible with the default parser regardless. The wealth of extra information obtained from linking `global` to the Universal Ctags parser is ultimately discarded anyway, yet parsing may take considerably longer. If this is the case, "default" behaviour may be recovered by first running the following command:

```
Sys.unsetenv (c ("GTAGSCONF", "GTAGSLABEL"))
```

See information on how to install the plugin for more details.

3.5.2 Package Statistics

This vignette describes the statistics collated by `pkgstats`. The first section provides full descriptions of all data returned by the main `pkgstats()` function, and the second section describes the output of the `pkgstats_summary()` function which converts statistics to a single-row summary. Single-row summaries from multiple packages can be combined to represent the statistical properties of multiple packages in a single `data.frame` object. The `pkgstats()` function is applied to all CRAN packages on a regular basis, with results accessible with the `dl_pkgstats_data()` function.

Overview of Package Statistics

The main `pkgstats()` function returns a `list` of eight main components:

1. `"loc"` summarising "Lines of Code" in package sub-directories and languages;
2. `"vignettes"` containing counts of numbers of vignettes and demos;
3. `"data_stats"` summarising data files;
4. `"desc"` summarising the contents of the package "DESCRIPTION" file;
5. `"translations"` summarising translations into other (human) languages;
6. `"objects"`: a table of all "objects" in all languages;
7. `"network"`: a table of relationships between objects, such as function calls; and
8. `"external_calls"`: a detailed table of all calls made to all R functions.

The following sub-sections provide further detail on these components (except the simpler components of `"vignettes"`, `"data_stats"`, and `"translations"`). The results use the output of applying the function to the source code of this package:

```
s <- pkgstats () # run in root directory of `pkgstats` source
```

The result is a list of various data extracted from the code. All except for `objects` and `network` represent summary data:

```
s [!names (s) %in% c ("objects", "network", "external_calls")]
#> $loc
#> # A tibble: 4 × 11
#>   language dir       nfiles nlines ncode nempty nspaces nchars nexpr ntabs
#>   <chr>    <chr>      <int>  <int> <int>  <int>   <int>  <int> <int> <int>
#> 1 C++      src            3    364   276     67     932   6983     1     0
#> 2 R        R             24   4727   345    682     333 114334     1     0
#> 3 R        tests          7    300   234     61     511   5543     1     0
```

(continues on next page)

(continued from previous page)

```
#> 4 Rmd      vignettes      2    347   278     61    1483  11290     1     0
#> #  1 more variable: indentation <int>
#>
#> $vignettes
#> vignettes     demos
#>         2         0
#>
#> $data_stats
#>           n  total_size median_size
#>           0           0           0
#>
#> $desc
#>    package verion                     date license
#> 1 pkgstats  0.1.1 Tue Oct 17 14:04:27 2023   GPL-3
#>                                                                              ␣
↪urls
#> 1 https://docs.ropensci.org/pkgstats/,\nnhttps://github.com/ropensci-review-tools/
↪pkgstats
#>                                                    bugs aut ctb fnd rev ths
#> 1 https://github.com/ropensci-review-tools/pkgstats/issues   1   0   0   0   0
#>   trl depends                                                        imports
#> 1   0      NA brio, checkmate, dplyr, fs, igraph, methods, readr, sys, withr
#>                                                                              ␣
↪   suggests
#> 1 curl, hms, jsonlite, knitr, parallel, pkgbuild, Rcpp, rmarkdown, roxygen2, testthat,
↪ visNetwork
#>   enchances linking_to
#> 1      <NA>      cpp11
#>
#> $translations
#> [1] NA
```

These results demonstrate that many fields use `NA` to denote values of zero. The following sub-sections explore these various components generated by the `pkgstats()` function in more detail.

Lines of Code

The first item in the above list is "loc" for Lines-of-Code, which are counted using an internal routine specifically developed for R packages, and which provides more accurate and R-specific information than most open source code counting libraries. For example, the counts in `pkgstats` are able to distinguish and separately count code chunks and text lines in `.Rmd` files.

```
s$loc
#> # A tibble: 4 × 11
#>   language dir       nfiles nlines ncode nempty nspaces nchars nexpr ntabs
#>   <chr>    <chr>      <int>  <int> <int>  <int>   <int>  <int> <int> <int>
#> 1 C++      src            3    364   276     67     932   6983     1     0
#> 2 R        R             24   4727   345    682     333 114334     1     0
#> 3 R        tests          7    300   234     61     511   5543     1     0
#> 4 Rmd      vignettes      2    347   278     61    1483  11290     1     0
#> #  1 more variable: indentation <int>
```

That output includes the following components, grouped by both computer language and package directory:

1. `nfiles` = Numbers of files in each directory and language.
2. `nlines` = Total numbers of lines in all files.
3. `ncode` = Total numbers of lines of code.
4. `ndoc` = Total numbers of documentation or comment lines.
5. `nempty` = Total numbers of empty of blank lines.
6. `nspaces` = Total numbers of white spaces in all code lines, excluding leading indentation spaces.
7. `nchars` = Total numbers of non-white-space characters in all code lines.
8. `nexpr` = Median numbers of nested expressions in all lines which have any expressions (see below).
9. `ntabs` = Number of lines of code with initial tab indentation.
10. `indentation` = Number of spaces by which code is indented (with `-1` denoting tab-indentation).

Numbers of nested expressions are counted as numbers of brackets or braces of any type nested on a single line. The following line has one nested bracket:

```
x <- myfn ()
```

while the following has four:

```
x <- function () { return (myfn ()) }
```

Code with fewer nested expressions per line is generally easier to read, and this metric is provided as one indication of the general readability of code. A second relative indication may be extracted by converting numbers of spaces and characters to a measure of relative numbers of white spaces, noting that the `nchars` value quantifies total characters including white spaces.

```
index <- which (s$loc$dir %in% c ("R", "src")) # consider source code only
sum (s$loc$nspaces [index]) / sum (s$loc$nchars [index])
#> [1] 0.01042723
```

Finally, the `ntabs` statistic can be used to identify whether code uses tab characters as indentation, otherwise the `indentation` statistics indicate median numbers of white spaces by which code is indented. The `objects`, `network`, and `external_calls` items returned by the `pkgstats()` function are described further below.

"desc": The package "DESCRIPTION" file

The `desc` item looks like this:

```
s$desc
#>    package verion                     date license
#> 1 pkgstats  0.1.1 Tue Oct 17 14:04:27 2023   GPL-3
#>
↪urls
#> 1 https://docs.ropensci.org/pkgstats/,\nnhttps://github.com/ropensci-review-tools/
↪pkgstats
#>                                                bugs aut ctb fnd rev ths
#> 1 https://github.com/ropensci-review-tools/pkgstats/issues   1   0   0   0   0
#>   trl depends                                                        imports
#> 1   0      NA brio, checkmate, dplyr, fs, igraph, methods, readr, sys, withr
```

(continues on next page)

(continued from previous page)

```
#>                                                                                ␣
↪   suggests
#> 1 curl, hms, jsonlite, knitr, parallel, pkgbuild, Rcpp, rmarkdown, roxygen2, testthat,
↪ visNetwork
#>   enchances linking_to
#> 1      <NA>      cpp11
```

This item includes the following components:

- Package name, version, date, and license
- Package URL(s) (`urls`)
- URL for BugReports (`bugs`)
- Number of contributors with role of *author* (`desc_n_aut`), *contributor* (`desc_n_ctb`), *funder* (`desc_n_fnd`), *reviewer* (`desc_n_rev`), *thesis advisor* (`ths`), and *translator* (`trl`, relating to translation between computer and not spoken languages).
- Comma-separated character entries for all `depends`, `imports`, `suggests`, and `linking_to` packages.

The "Date" field is taken from the "Date/Publication" field automatically inserted by CRAN on package publication, or for non-CRAN packages to the "mtime" value (modification time) value of the DESCRIPTION file. Note that "date" values extracted by `pkgstats` do not use "Date" values from DESCRIPTION files (as these are manually-entered, and potentially unreliable).

1.3 "objects": Objects in all languages

The `objects` item contains all code objects identified by the code-tagging library `ctags`. For R, those are primarily functions, but for other languages may be a variety of entities such as class or structure definitions, or sub-members thereof. Object tables look like this:

```
head (s$objects)
#>           file_name                   fn_name     kind language loc npars
#> 1 R/archive-trawl.R     pkgstats_from_archive function        R  89     9
#> 2 R/archive-trawl.R        list_archive_files function        R  17     2
#> 3 R/archive-trawl.R             rm_prev_files function        R  24     2
#> 4 R/archive-trawl.R pkgstats_fns_from_archive function        R  82     7
#> 5         R/cpp11.R                  cpp_loc function        R   3     4
#> 6 R/ctags-install.R               clone_ctag function        R  17     1
#>   has_dots exported param_nchards_md param_nchards_mn num_doclines
#> 1    FALSE     TRUE              133         159.7778           77
#> 2    FALSE    FALSE               NA               NA           NA
#> 3    FALSE    FALSE               NA               NA           NA
#> 4    FALSE     TRUE              163         174.5714           50
#> 5    FALSE    FALSE               NA               NA           NA
#> 6    FALSE    FALSE               NA               NA           NA
```

Objects are primarily sorted by language, with R-language objects given first. These are mostly functions, and include statistics on:

- lines of code used to define each function (`loc`);
- numbers of parameters (`npars`);
- whether or not the function includes a "three dots" parameter (that is, `...`; identified by `has_dots`);

- whether or not a function is exported (`exported`);
- Mean and median numbers of character used to document each parameter (`param_nchards_mn` and `param_nchards_md`, respectively); and
- Total number of lines of documentation for that object / function.

"network": Relationships between objects

The `network` item details all relationships between objects, which generally reflects one object calling or otherwise depending on another object. Each row thus represents one edge of a "function call" network, with each entry in the `from` and `to` columns representing the network vertices or nodes.

```
head (s$network)
#>                   file line1                    from                         to
#> 1   R/external_calls.R    11   external_call_network      extract_call_content
#> 2   R/external_calls.R    26   external_call_network add_base_recommended_pkgs
#> 3   R/external_calls.R    38   external_call_network   add_other_pkgs_to_calls
#> 4   R/external_calls.R   326 add_other_pkgs_to_calls             control_parse
#> 5 R/pkgstats-summary.R    39        pkgstats_summary                null_stats
#> 6 R/pkgstats-summary.R    50        pkgstats_summary               loc_summary
#>   language cluster_dir centrality_dir cluster_undir centrality_undir
#> 1        R           1              9             1         230.8333
#> 2        R           1              9             1         230.8333
#> 3        R           1              9             1         230.8333
#> 4        R           1              1             1           6.0000
#> 5        R           1             11             1         874.0000
#> 6        R           1             11             1         874.0000
nrow (s$network)
#> [1] 142
```

The network table includes additional statistics on the centrality of each edge, measured as betweenness centrality assuming edges to be both directed (`centrality_dir`) and undirected (`centrality_undir`). More central edges reflect connections between objects that are more central to package functionality, and vice versa. The distinct components of the network are also represented by discrete cluster numbers, calculated both for directed and undirected versions of the network. Each distinct cluster number represents a distinct group of objects, internally related to other members of the same cluster, yet independent of all objects with different cluster numbers.

The network can be viewed as an interactive `vis.js` network through passing the result of `pkgstats` – the variable `p` in the code above – to the `plot_network()` function.

"external_calls": All calls made to all R functions

The `external_calls` item is structured similar to the `network` object, but identifies all calls to functions from external packages. However, unlike the `network` and `object` data, which provide information on objects and relationships in all computer languages used within a package, the `external_calls` object maps calls within R code only, in order to provide insight into the use within a package of of functions from other packages, including R's base and recommended packages. The object looks like this:

```
head (s$external_calls)
#>   tags_line      call        tag           file     kind start end package
#> 1         1         c GTAGSLABEL R/ctags-test.R nameattr   109 109    base
#> 2         2 character  file_name   R/pkgstats.R nameattr   185 185    base
#> 3         3 character    fn_name   R/pkgstats.R nameattr   186 186    base
```

(continues on next page)

(continued from previous page)

```
#> 4         4   logical   has_dots   R/pkgstats.R nameattr   189 189    base
#> 5         5   integer        loc   R/pkgstats.R nameattr   187 187    base
#> 6         6 left_join       name       R/plot.R nameattr    89  89   dplyr
```

These data are converted to a summary form by the `pkgstats_summary()` function, which tabulates numbers of external calls and unique functions from each package. These data are presented as a single character string which looks like this:

```
s_summ <- pkgstats_summary (s)
print (s_summ$external_calls)
```

These data can be easily converted to the corresponding numeric values using code like the following:

```
x <- strsplit (s_summ$external_calls, ",") [[1]]
x <- do.call (rbind, strsplit (x, ":"))
x <- data.frame (
    pkg = x [, 1],
    n_total = as.integer (x [, 2]),
    n_unique = as.integer (x [, 3])
)
x$n_total_rel <- round (x$n_total / sum (x$n_total), 3)
x$n_unique_rel <- round (x$n_unique / sum (x$n_unique), 3)
print (x)
#>           pkg n_total n_unique n_total_rel n_unique_rel
#> 1        base     581       84       0.704        0.426
#> 2        brio      11        2       0.013        0.010
#> 3        curl       4        3       0.005        0.015
#> 4       dplyr       7        4       0.008        0.020
#> 5          fs       4        2       0.005        0.010
#> 6    graphics      10        2       0.012        0.010
#> 7         hms       2        1       0.002        0.005
#> 8      igraph       3        3       0.004        0.015
#> 9    parallel       2        1       0.002        0.005
#> 10   pkgstats     126       73       0.153        0.371
#> 11      readr       8        5       0.010        0.025
#> 12      stats      19        3       0.023        0.015
#> 13        sys      14        1       0.017        0.005
#> 14      tools       3        2       0.004        0.010
#> 15      utils      22        7       0.027        0.036
#> 16 visNetwork       3        2       0.004        0.010
#> 17      withr       6        2       0.007        0.010
```

Those data reveal, for example, that this package makes 581 individual calls to 84 unique functions from the "base" package.

CHAPTER

FOUR

PKGCHECK

Check whether a package is ready for submission to rOpenSci's peer review system. The primary function collates the output of `goodpractice`, including `R CMD check` results, a number of statistics via the `pkgstats` package, and checks for package structure expected for rOpenSci submissions. The output of this function immediately indicates whether or not a package is "Ready to Submit".

4.1 Installation

The easiest way to install this package is via the associated `r-universe`. As shown there, simply enable the universe with

```
options (repos = c (
    ropenscireviewtools = "https://ropensci-review-tools.r-universe.dev",
    CRAN = "https://cloud.r-project.org"
))
```

And then install the usual way with,

```
install.packages ("pkgcheck")
```

Alternatively, the package can be installed by first installing either the remotes or pak packages and running one of the following lines:

```
remotes::install_github ("ropensci-review-tools/pkgcheck")
pak::pkg_install ("ropensci-review-tools/pkgcheck")
```

The package can then loaded for use with

```
library (pkgcheck)
```

4.2 Setup

The pkgstats package also requires the system libraries ctags and GNU global to be installed. Procedures to install these libraries on various operating systems are described in a pkgstats vignette. This package also uses the GitHub GraphQL API which requires a local GitHub token to be stored with an unambiguous name including GITHUB, such as GITHUB_TOKEN (recommended), or GITHUB_PAT (for Personal Authorization Token). This can be obtained from GitHub (via your user settings), and stored using

```
Sys.setenv ("GITHUB_TOKEN" = "<my_token>")
```

This can also be set permanently by putting this line in your ~/.Renviron file (or creating this if it does not yet exist). Once pkgstats has been successfully installed, the pkgcheck package can then be loaded via a library call:

```
library (pkgcheck)
```

4.3 Usage

The package primarily has one function, pkgcheck, which accepts the single argument, path, specifying the local location of a git repository to be analysed. The following code generates a reproducible report by first downloading a local clone of a repository called srr-demo, which contains the skeleton of an srr (Software Review Roclets) package, generated with the srr_stats_pkg_skeleton() function:

```
mydir <- file.path (tempdir (), "srr-demo")
gert::git_clone ("https://github.com/mpadge/srr-demo", path = mydir)
x <- pkgcheck (mydir)
```

That object has default print and summary methods. The latter can be used to simply check whether a package is ready for submission:

```
summary (x)
##
## ── demo 0.0.0.9000 ──────────────────────────────────────────────────────────────
##
## ✓ Package name is available
## ✖ does not have a 'codemeta.json' file.
## ✖ does not have a 'contributing' file.
## ✓ uses 'roxygen2'.
## ✓ 'DESCRIPTION' has a URL field.
## ✖ 'DESCRIPTION' does not have a BugReports field.
## ✖ Package has no HTML vignettes
## ✓ All functions have examples.
## ✖ Package has no continuous integration checks.
## ✖ Package coverage failed
## ✓ R CMD check found no errors.
## ✓ R CMD check found no warnings.
##
##  Current status:
## ✖ This package is not ready to be submitted.
##
```

A package may only be submitted when the summary contains all ticks and no cross symbols. (These symbols are colour-coded with green ticks and red crosses when generated in a terminal; GitHub markdown only renders them in black-and-white.) The object returned from the `pkgcheck` function is a complex nested list with around a dozen primary components. Full information can be obtained by simply calling the default `print` method by typing the object name (`x`).

4.4 The pkgcheck GitHub action

The `pkgcheck` package also has an associated GitHub action in the `pkgcheck-action` repository. You can use this action to run `pkgcheck` every time you push commits to GitHub, just like the `rcmdcheck()` checks which can be installed and run via the `usethis::use_github_action_check_standard()` function. `pkgcheck` includes an analogous function, `use_github_action_pkgcheck()`, which will download the workflow file defining the action into your local `.github/workflows` folder. See the `pkgcheck-action` repository for more details.

You can also add a `pkgcheck` badge, just like that `rcmdcheck` badge, that will always reflect the current state of your repository's `pkgcheck` results. To add a badge, copy the following line to your README file, filling in details of the GitHub organization and repository name (`<org>` and `<repo>`, respectively):

```
[![pkgcheck](https://github.com/<org>/<repo>/workflows/pkgcheck/badge.svg)](https://
↪github.com/<org>/<repo>/actions?query=workflow%3Apkgcheck)
```

4.5 What is checked?

All current checks are listed in a separate vignette.

4.5.1 Summary of Check Results

Calling `summary()` on the object returned by the `pkgcheck()` function will generate a checklist like that shown above. This checklist will also be automatically generated when a package is first submitted to rOpenSci, and is used by the editors to assess whether to process a submission. Authors must ensure prior to submission that there are no red crosses in the resultant list. (In the unlikely circumstances that a package is unable to pass particular checks, explanations should be given upon submission about why those checks fail, and why review may proceed in spite of such failures.)

4.5.2 Detailed Check Results

Full details of check results can be seen by `print`-ing the object returned by the `pkgcheck()` function (or just by typing the name of this object in the console.)

The package includes an additional function, `checks_to_markdown()`, with a parameter, `render`, which can be set to `TRUE` to automatically render a HTML-formatted representation of the check results, and open it in a browser. The formatting differs only slightly from the terminal output, mostly through the components of `goodpractice` being divided into distinct headings, with explicit statements in cases where components pass all checks (the default screen output inherits directly from that package, and only reports components which *do not* pass all checks).

This `checks_to_markdown()` function returns the report in markdown format, suitable for pasting directly into a GitHub issue, or other markdown-compatible place. (The `clipr` package can be used to copy this directly to your local clipboard with `write_clip(md)`, where `md` is the output of `checks_to_markdown()`.)

4.6 Caching and running pkgcheck in the background

Running the pgkcheck function can be time-consuming, primarily because the goodpractice component runs both a full R CMD check, and calculates code coverage of all tests. To avoid re-generating these results each time, the package saves previous reports to a local cache directory defined in Sys.getenv("PKGCHECK_CACHE_DIR").

You may manually erase the contents of this pkgcheck subdirectory at any time at no risk beyond additional time required to re-generate contents. By default checks presume packages use git for version control, with checks updated only when code is updated via git commit. Checks for packages that do not use git are updated when any files are modified.

The first time pkgcheck() is applied to a package, the checks will be stored in the cache directory. Calling that function a second time will then load the cached results, and so enable checks to be returned much faster. For code which is frequently updated, such as for packages working on the final stages prior to submission, it may still be necessary to repeatedly call pkgcheck() after each modification, a step which may still be inconveniently time-consuming. To facilitate frequent re-checking, the package also has a pkgcheck_bg() function which is effectively identical to the main pkgcheck() function, except it runs in the background, enabling you to continue coding while checks are running.

The pkgcheck_bg() function returns a handle to the callr::r_bg() process in which the checks are running. Typing the name of the returned object will immediately indicate whether the checks are still running, or whether they have finished. That handle is itself an R6 object with a number of methods, notably including get_result() which can be used to access the checks once the process has finished. Alternatively, as soon as the background process, the normal (foreground) pkgcheck() function may be called to quickly re-load the cached results.

4.7 Prior Work

The checklist package for "checking packages and R code".

4.8 Code of Conduct

Please note that this package is released with a Contributor Code of Conduct. By contributing to this project, you agree to abide by its terms.

4.9 Contributors

All contributions to this project are gratefully acknowledged using the allcontributors package following the all-contributors specification. Contributions of any kind are welcome!

4.9.1 Code

4.9.2 Issue Authors

4.9.3 Issue Contributors

4.10 Functions

4.10.1 checks_to_markdown

Convert checks to markdown-formatted report

Description

Convert checks to markdown-formatted report

Usage

```
checks_to_markdown(checks, render = FALSE)
```

Arguments

Argument	Description
checks	Result of main *pkgcheck* function
render	If TRUE , render output as html document and open in browser.

Value

Markdown-formatted version of check report

Seealso

Other extra: *list_pkgchecks* , *logfile_names* , *render_md2html*

Examples

```
checks <- pkgcheck ("/path/to/my/package")
md <- checks_to_markdown (checks) # markdown-formatted character vector
md <- checks_to_markdown (checks, render = TRUE) # HTML version
```

4.10.2 get_default_github_branch

get_default_github_branch

Description

get_default_github_branch

Usage

```
get_default_github_branch(org, repo)
```

Arguments

Argument	Description
org	Github organization
repo	Github repository

Value

Name of default branch on GitHub

Seealso

Other github: *get_gh_token* , *get_latest_commit* , *use_github_action_pkgcheck*

Note

This function is not intended to be called directly, and is only exported to enable it to be used within the plumber API.

Examples

```
org <- "ropensci-review-tools"
repo <- "pkgcheck"
branch <- get_default_github_branch (org, repo)
```

4.10.3 get_gh_token

Get GitHub token

Description

Get GitHub token

Usage

```
get_gh_token(token_name = "")
```

Arguments

Argument	Description
token_name	Optional name of token to use

Value

The value of the GitHub access token extracted from environment variables.

Seealso

Other github: *get_default_github_branch* , *get_latest_commit* , *use_github_action_pkgcheck*

Examples

```
token <- get_gh_token ()
```

4.10.4 get_latest_commit

get_latest_commit

Description

get_latest_commit

Usage

```
get_latest_commit(org, repo, branch = NULL)
```

Arguments

Argument	Description
org	Github organization
repo	Github repository
branch	Branch from which to get latest commit

Value

Details of latest commit including OID hash

Seealso

Other github: *get_default_github_branch* , *get_gh_token* , *use_github_action_pkgcheck*

Note

This returns the latest commit from the default branch as specified on GitHub, which will not necessarily be the same as information returned from `gert::git_info` if the `HEAD` of a local repository does not point to the same default branch.

Examples

```
org <- "ropensci-review-tools"
repo <- "pkgcheck"
commit <- get_latest_commit (org, repo)
```

4.10.5 list_pkgchecks

List all checks currently implemented

Description

List all checks currently implemented

Usage

```
list_pkgchecks(quiet = FALSE)
```

Arguments

Argument	Description
quiet	If TRUE , print all checks to screen. Function invisibly returns list of checks regardless.

Value

Character vector of names of all checks (invisibly)

Seealso

Other extra: *checks_to_markdown* , *logfile_names* , *render_md2html*

Examples

```
list_pkgchecks ()
```

4.10.6 logfile_names

Set up stdout & stderr cache files for r_bg process

Description

Set up stdout & stderr cache files for r_bg process

Usage

```
logfile_names(path)
```

Arguments

Argument	Description
path	Path to local repository

Value

Vector of two strings holding respective local paths to `stdout` and `stderr` files for `r_bg` process controlling the main *pkgcheck* function when executed in background mode.

Seealso

Other extra: *checks_to_markdown* , *list_pkgchecks* , *render_md2html*

Note

These files are needed for the callr `r_bg` process which controls the main *pkgcheck* . The `stdout` and `stderr` pipes from the process are stored in the cache directory so they can be inspected via their own distinct endpoint calls.

Examples

```
logfiles <- logfiles_namnes ("/path/to/my/package")
print (logfiles)
```

4.10.7 pkgcheck_bg

Generate report on package compliance with rOpenSci Statistical Software requirements as background process

Description

Generate report on package compliance with rOpenSci Statistical Software requirements as background process

Usage

```
pkgcheck_bg(path)
```

Arguments

Argument	Description
`path`	Path to local repository

Value

A processx object connecting to the background process generating the main *pkgcheck* results (see Note).

Seealso

Other pkgcheck_fns: *pkgcheck* , *print.pkgcheck*

Note

The return object will by default display whether it is still running, or whether it has finished. Once it has finished, the results can be obtained by calling $get_result() , or the main *pkgcheck* function can be called to quickly retrieve the main results from local cache.

This function does not accept the `extra_env` parameter of the main *pkgcheck* function, and can not be used to run extra, locally-defined checks.

Examples

```
# Foreground checks as "blocking" process which will return
# only after all checks have finished:
checks <- pkgcheck ("/path/to/my/package")

# Or run process in background, do other things in the meantime,
# and obtain checks once they have finished:
ps <- pkgcheck_bg ("/path/to/my/package")
ps # print status to screen, same as 'ps$print()'
# To examine process state while running:
f <- ps$get_output_file ()
readLines (f) # or directly open file with local file viewer
# ... ultimately wait until 'running' changes to 'finished', then:
checks <- ps$get_result ()
```

4.10.8 pkgcheck-package

pkgcheck: rOpenSci Package Checks

Description

Check whether a package is ready for submission to rOpenSci's peer review system.

Seealso

Useful links:

- https://docs.ropensci.org/pkgcheck/
- https://github.com/ropensci-review-tools/pkgcheck
- Report bugs at https://github.com/ropensci-review-tools/pkgcheck/issues

Author

Maintainer : Mark Padgham mark.padgham@email.com (ORCID)

Authors:

- Maëlle Salmon
- Jacob Wujciak-Jens jacob@wujciak.de (ORCID)

4.10.9 pkgcheck

Generate report on package compliance with rOpenSci Statistical Software requirements

Description

Generate report on package compliance with rOpenSci Statistical Software requirements

Usage

```
pkgcheck(
  path = ".",
  goodpractice = TRUE,
  use_cache = TRUE,
  extra_env = .GlobalEnv
)
```

Arguments

Argument	Description
path	Path to local repository
goodprac	If FALSE , skip goodpractice checks. May be useful in development stages to more quickly check other aspects.
use_cach	Checks are cached for rapid retrieval, and only re-run if the git hash of the local repository changes. Setting use_cache to FALSE will for checks to be re-run even if the git hash has not changed.
extra_er	Additional environments from which to collate checks. Other package names may be appended using c , as in c(.GlobalEnv, "mypkg") .

Value

A pkgcheck object detailing all package assessments automatically applied to packages submitted for peer review.

Seealso

Other pkgcheck_fns: *pkgcheck_bg* , *print.pkgcheck*

Examples

```
checks <- pkgcheck ("/path/to/my/package") # default full check
summary (checks)
# Or to run only checks implemented in 'pkgcheck' and not the
# additional \pkg{goodpractice} checks:
checks <- pkgcheck ("/path/to/my/package", goodpractice = FALSE)
summary (checks)
```

4.10.10 pkgstats_data

Statistics on all CRAN packages from 'pkgstats'

Description

Statistics on all CRAN packages from 'pkgstats'

Format

An object of class grouped_df (inherits from tbl_df , tbl , data.frame) with 21091 rows and 91 columns.

Usage

```
pkgstats_data
```

4.10.11 print.pkgcheck

Generic print method for 'pkgcheck' objects.

Description

Generic print method for 'pkgcheck' objects.

Usage

```
list(list("print"), list("pkgcheck"))(x, deps = FALSE, ...)
```

Arguments

Argument	Description
x	A 'pkgcheck' object to be printed.
deps	If 'TRUE', include details of dependency packages and function usage.
...	Further arguments pass to or from other methods (not used here).

Value

Nothing. Method called purely for side-effect of printing to screen.

Seealso

Other pkgcheck_fns: *pkgcheck_bg* , *pkgcheck*

Examples

```
checks <- pkgcheck ("/path/to/my/package")
print (checks) # print full checks, starting with summary
summary (checks) # print summary only
```

4.10.12 read_pkg_guide

Browse packaging guidelines

Description

A convenience function to automatically open the web page of rOpenSci's "Package Development Guide" in the default browser.

Usage

```
read_pkg_guide(which = c("release", "dev"))
```

Arguments

Argument	Description
which	Whether to read the released or "dev" development version.

Value

Nothing. Function called purely for side-effect of opening web page with package guidelines.

Examples

```
read_pkg_guide ()
```

4.10.13 render_md2html

render markdown-formatted input into 'html'

Description

render markdown-formatted input into 'html'

Usage

```
render_md2html(md, open = TRUE)
```

Arguments

Argument	Description
md	Result of *checks_to_markdown* function.
open	If TRUE , open hmtl -rendered version in web browser.

Value

(invisible) Location of `.html` -formatted version of input.

Seealso

Other extra: *checks_to_markdown* , *list_pkgchecks* , *logfile_names*

Examples

```
checks <- pkgcheck ("/path/to/my/package")
# Generate standard markdown-formatted character vector:
md <- checks_to_markdown (checks)

# Directly generate HTML output:
h <- checks_to_markdown (checks, render = TRUE) # HTML version

# Or convert markdown-formatted version to HTML:
h <- render_md2html (md)
```

4.10.14 use_github_action_pkgcheck

Use pkgcheck Github Action

Description

Creates a Github workflow file in `dir` integrate *pkgcheck()* into your CI.

Usage

```
use_github_action_pkgcheck(
  dir = ".github/workflows",
  overwrite = FALSE,
  file_name = "pkgcheck.yaml",
  branch = gert::git_branch(),
  inputs = NULL
)
```

Arguments

Argument	Description
`dir`	Directory the file is written to.
`overwrite`	Overwrite existing file?
`file_name`	Name of the workflow file.
`branch`	Name of git branch for checks to run; defaults to currently active branch.
`inputs`	Named list of inputs to the `ropensci-review-tools/pkgcheck-action` . See details below.

Details

For more information on the action and advanced usage visit the action repository .

Value

The path to the new file, invisibly.

Seealso

Other github: *get_default_github_branch* , *get_gh_token* , *get_latest_commit*

Examples

```
use_github_action_pkgcheck (inputs = list (`post-to-issue` = "false"))
use_github_action_pkgcheck (branch = "main")
```

4.11 Vignettes

4.11.1 Extending or modifying checks

This vignette describes how to modify or extend the existing suite of checks implemented by `pkgcheck`. Each of the internal checks is defined in a separate file in the `R` directory of this package with the prefix of `check_` (or `checks_` for files which define multiple, related checks). Checks only require two main functions, the first defining the check itself, and the second defining `summary` and `print` methods based on the result of the first function. The check functions must have a prefix `pkgchk_`, and the second functions defining output methods specifying must have a prefix `output_pkgchk_`. These two kind of function are now described in the following two sections.

Both of these functions must also accept a single input parameter of a `pkgcheck` object, by convention named `checks`. This object is a list of four main items:

1. `pkg` which summarises data extracted from `pkgstats::pkgstats()`, and includes essential information on the package being checked.
2. `info` which contains information used in checks, including `info$git` detailing git repository information, `info$pkgstats` containing a summary of a few statistics generated from `pkgstats::pkgstats()`, along with statistical comparisons against distributions from all current CRAN packages, an `info$network_file` specifying a local directory to a `vis.js` visualisation of the function call network of the package, and an `info$badges` item containing information from GitHub workflows and associated badges, where available.
3. `checks` which contains a list of all objects returned from all `pkgchk_...()` functions, which are used as input to `output_pkgchk_...()` functions.
4. `meta` containing a named character vector of versions of the core packages used in `pkgcheck`.

`pkgcheck` objects generally also include a fifth item, `goodpractice`, containing the results of `goodpractice` checks. The `checks` item passed to each `pkgchk_...()` function contains all information on the `package`, `info`, `meta`, and (optionally) `goodpractice` items. Checks may use any of this information, or even add additional information as demonstrated below. The `checks$checks` list represents the output of check functions, and may not be used in any way within `pkgchk_...()` functions.

This is the output of applying `pkgcheck` to a package generated with the `srr` function `srr_stats_pkg_skeleton()`, with `goodpractice = FALSE` to suppress that part of the results.

```
#> List of 4
#>  $ pkg   :List of 8
#>   ..$ name        : chr "dummypkg"
#>   ..$ path        : chr "/tmp/RtmpkguwJc/dummypkg"
#>   ..$ version     : chr "0.0.0.9000"
#>   ..$ url         : chr(0)
#>   ..$ BugReports  : chr(0)
#>   ..$ license     : chr "GPL-3"
#>   ..$ summary     :List of 12
#>   .. ..$ num_authors         : int 1
#>   .. ..$ num_vignettes       : int 0
#>   .. ..$ num_data            : int 0
#>   .. ..$ imported_pkgs       : int 1
#>   .. ..$ num_exported_fns    : int 1
#>   .. ..$ num_non_exported_fns: int 2
#>   .. ..$ num_src_fns         : int 2
#>   .. ..$ loc_exported_fns    : int 3
#>   .. ..$ loc_non_exported_fns: int 3
#>   .. ..$ loc_src_fns         : int 5
#>   .. ..$ num_params_per_fn   : int 0
#>   .. ..$ languages           : chr [1:2] "C++: 72%" "R: 28%"
#>   ..$ dependencies:'data.frame': 4 obs. of  2 variables:
#>   .. ..$ type   : chr [1:4] "depends" "imports" "suggests" "linking_to"
#>   .. ..$ package: chr [1:4] "NA" "Rcpp" "testthat" "Rcpp"
#>  $ info  :List of 5
#>   ..$ git         : list()
#>   ..$ srr         :List of 5
#>   .. ..$ message     : chr [1:108] "This package still has TODO standards and can not␣
↪be submitted" "Package can not be submitted because the following standards [v0.1.0]␣
↪are missing from your code:" "" "G1.0" ...
#>   .. ..$ categories  : chr "Regression and Supervised Learning"
#>   .. ..$ missing_stds: chr "G1.0, G1.4a, G1.6, G2.0a, G2.1a, G2.2, G2.3a, G2.3b, G2.4,
↪ G2.4a, G2.4b, G2.4c, G2.4d, G2.4e, G2.5, G2.6, G2.7,"| __truncated__
#>   .. ..$ report_file : chr "/home/smexus/.cache/pkgcheck/static/dummypkg_srr2021-10-
↪15-16:46:34.html"
#>   .. ..$ okay        : logi FALSE
#>   ..$ pkgstats    :'data.frame': 25 obs. of  4 variables:
#>   .. ..$ measure   : chr [1:25] "files_R" "files_src" "files_vignettes" "files_tests"␣
↪...
#>   .. ..$ value     : num [1:25] 4 2 0 2 10 26 6 0 3 1 ...
#>   .. ..$ percentile: num [1:25] 23.284 77.356 0 64.15 0.445 ...
#>   .. ..$ noteworthy: chr [1:25] "" "" "TRUE" "" ...
#>   .. ..- attr(*, "language")= chr [1:2] "C++: 72%" "R: 28%"
#>   .. ..- attr(*, "files")= chr [1:2] "C++: 2" "R: 4"
#>   ..$ network_file: chr "/home/smexus/.cache/pkgcheck/static/dummypkg_pkgstats.html"
#>   ..$ badges      : list()
#>  $ checks:List of 12
#>   ..$ fns_have_exs     : Named logi FALSE
#>   .. ..- attr(*, "names")= chr "test_fn.Rd"
#>   ..$ has_bugs         : logi FALSE
#>   ..$ has_citation     : logi FALSE
#>   ..$ has_codemeta     : logi FALSE
#>   ..$ has_contrib_md   : logi FALSE
```

(continues on next page)

(continued from previous page)

```
#>   ..$ has_scrap        : chr(0)
#>   ..$ has_url          : logi FALSE
#>   ..$ has_vignette     : logi FALSE
#>   ..$ left_assign      :List of 2
#>   .. ..$ global: logi FALSE
#>   .. ..$ usage : Named num [1:2] 2 0
#>   .. .. ..- attr(*, "names")= chr [1:2] "<-" "="
#>   ..$ on_cran          : logi FALSE
#>   ..$ pkgname_available: logi TRUE
#>   ..$ uses_roxygen2    : logi TRUE
#>  $ meta  : Named chr [1:3] "0.0.2.25" "0.0.2.96" "0.0.1.120"
#>   ..- attr(*, "names")= chr [1:3] "pkgstats" "pkgcheck" "srr"
#>  - attr(*, "class")= chr [1:2] "pkgcheck" "list"
#> NULL
```

1. The check function

An example is the check for whether a package has a citation, defined in `R/check_has_citation.R`:

```
#' Check whether a package has a `inst/CITATION` file.
#'
#' "CITATION" files are required for all rOpenSci packages, as documented [in
#' our "*Packaging
#' Guide*](https://devguide.ropensci.org/building.html#citation-file). This does
#' not check the contents of that file in any way.
#'
#' @param checks A 'pkgcheck' object with full \pkg{pkgstats} summary and
#' \pkg{goodpractice} results.
#' @noRd
pkgchk_has_citation <- function (checks) {

    "CITATION" %in% list.files (fs::path (checks$pkg$path, "inst"))
}
```

This check is particularly simple, because a `"CITATION"` file must have exactly that name, and must be in the `inst` sub-directory. This function returns a simple logical of `TRUE` if the expected `"CITATION"` file is present, otherwise it returns `FALSE`. This function, and all functions beginning with the prefix `pkgchk_`, will be automatically called by the main `pkgcheck()` function, and the value stored in `checks$checks$has_citation`. The name of the item within the `checks$checks` list is the name of the function with the `pkgchk_` prefix removed.

A more complicated example is the function to check whether a package contains files which should not be there – internally called "scrap" files. The check function itself, defined in `R/check-scrap.R`, checks for the presence of files matching an internally-defined list including files used to locally cache folder thumbnails such as `".DS_Store"` or `"Thumbs.db"`. The function returns a character vector of the names of any "scrap" files which can be used by the `print` method to provide details of files which should be removed. This illustrates the first general principle of these check functions; that,

- *Any information needed when summarising or printing the check result should be returned from the main check function.*

A second important principle is that,

- *Check functions should never return `NULL`, rather should always return an empty vector (such as `integer(0)`).*

The following section considers how these return values from check functions are converted to `summary` and `print` output.

2. The output function

All `output_pkgchk_...()` functions must also accept the single input parameter of `checks`, in which the `checks$checks` sub-list will already have been populated by calling all `pkgchk_...()` functions described in the previous section. The `pkgchk_has_citation()` function will create an entry of `checks$checks$has_citation` which contains the binary flag indicating whether or not a `"CITATION"` file is present. Similarly, the the `pkgchk_has_scrap()` function will create `checks$checks$has_scrap` which will contain names of any scrap files present, and a length-zero vector otherwise.

The `output_pkgchk_has_citation()` function then looks like this:

```
output_pkgchk_has_citation <- function (checks) {

    out <- list (
        check_pass = checks$checks$has_citation,
        summary = "",
        print = ""
    )

    # disabled:
    # https://github.com/ropensci-review-tools/pkgcheck/issues/115
    # out$summary <- paste0 (
    #    ifelse (out$check_pass, "has", "does not have"),
    #    " a 'CITATION' file."
    # )

    return (out)
}
```

The first lines are common to all `output_pkgchk_...()` functions, and define the generic return object. This object must be a list with the following three items:

1. `check_pass` as binary flag indicating whether or not a check was passed;
2. `summary` containing text used to generate the `summary` output; and
3. `print` containing information used to generate the `print` output, itself a `list` of the following items:
    - A `msg_pre` to display at the start of the `print` result;
    - An `object` to be printed, such as a vector of values, or a `data.frame`.
    - A `msg_post` to display at the end of the `print` result following the `object`.

`summary` and `print` methods may be suppressed by assigning values of `""`. The above example of `pkgcheck_has_citation` has `print = ""`, and so no information from this check will appear as output of the `print` method. The `summary` field is commented-out in the current version, but left to illustrate here that it has a value that is specified for both `TRUE` and `FALSE` values of `check_pass`, via an `ifelse` statement. The value is determined by the result of the main `pkgchk_has_citation()` call, and is converted into a green tick if `TRUE`, or a red cross if `FALSE`.

Checks for which `print` information is desired require a non-empty `print` item, as in the `output_pkgchk_has_scrap()` function:

```
output_pkgchk_has_scrap <- function (checks) {

    out <- list (
        check_pass = length (checks$checks$has_scrap) == 0L,
        summary = "",
        print = ""
    )

    if (!out$check_pass) {
        out$summary <- "Package contains unexpected files."
        out$print <- list (
            msg_pre = paste0 (
                "Package contains the ",
                "following unexpected files:"
            ),
            obj = checks$checks$has_scrap,
            msg_post = character (0)
        )
    }

    return (out)
}
```

In this case, both `summary` and `print` methods are only triggered `if (!out$check_pass)` – so only if the check fails. The `print` method generates the heading specified in `out$print$msg_pre`, with any vector-valued objects stored in the corresponding `obj` list item displayed as formatted lists. A package with "scrap" files, `"a"` and `"b"`, would thus have `out$print$obj <- c ("a", "b")`, and when printed would look like this:

```
#> ✖ Package contains the following unexpected files:
#> • a
#> • b
```

This formatting is also translated into corresponding markdown and HTML formatting in the `checks_to_markdown()` function.

The design of these `pkgchk_` and `output_pkgchk_` functions aims to make the package readily extensible, and we welcome discussions about developing new checks. The primary criterion for new package-internal checks is that they must be of very general applicability, in that they should check for a condition that *almost* every package should or should not meet.

The package also has a mechanism to easily incorporate more specific, locally-defined checks, as explored in the following section.

3. Creating new checks

3.1 New Local Checks (*for package users*)

The main `pkgcheck()` function has an additional parameter, `extra_env` which specifies,

> Additional environments from which to collate checks. Other package names may be appended using c, as in c(.GlobalEnv, "mypkg").

This allows specific checks to be defined locally, and run by passing the name of the environment in which those checks are defined in this parameter. This section illustrates the process using the bundled "tarball" (that is, `.tar.gz` file) of one version of the `pkgstats` package included with that package.

```
f <- system.file ("extdata", "pkgstats_9.9.tar.gz", package = "pkgstats")
path <- pkgstats::extract_tarball (f)
checks <- pkgcheck (path)
summary (checks)
```

```
#>
#> ── pkgstats 9.9␣
↪──────────────────────────────────────────────────────────────────────────────
#>
#> ✓ Package name is available
#> ✖ does not have a 'codemeta.json' file.
#> ✖ does not have a 'contributing' file.
#> ✓ uses 'roxygen2'.
#> ✓ 'DESCRIPTION' has a URL field.
#> ✓ 'DESCRIPTION' has a BugReports field.
#> ✖ Package has no HTML vignettes
#> ✖ These functions do not have examples: [pkgstats_from_archive].
#> ✓ Package has continuous integration checks.
#> ✖ Package coverage failed
#> ✖ R CMD check found 1 error.
#> ✓ R CMD check found no warnings.
#>
#>  Current status:
#> ✖ This package is not ready to be submitted.
```

Let's now presume I have a reputation in the R community for all of my packages starting with "aa", to ensure they are always listed first. This section demonstrates how to implement a check that only passes if the first two letters of the package name are "aa". The first step described above is to define the check itself via a function prefixed with `pkgchk_`. The easiest approach would be for the `pkgcheck_` function to directly check the name, and return a logical flag indicating whether or not the same starts with "aa". The resultant `summary` and `print` methods can, however, only use the information provided by the initial `pkgchk_` function. That means if we want to print the actual name in the result of either of those functions, to show that it indeed does not form the desired patter, we need to return that information. The check function is then simply:

```
pkgchk_starts_with_aa <- function (checks) {
    checks$pkg$name
}
```

We then need to define the output functions:

```
output_pkgchk_starts_with_aa <- function (checks) {

    out <- list (
                 check_pass = grepl ("^aa",
                                     checks$checks$starts_with_aa,
                                     ignore.case = TRUE),
                 summary = "",
                 print = ""
    )

    out$summary <- paste0 ("Package name [",
                           checks$checks$starts_with_aa,
                           "] does ",
```

(continues on next page)

(continued from previous page)

```
                            ifelse (out$check_pass,
                                    "",
                                    "NOT"),
                            " start with 'aa'")

    return (out)
}
```

If we simply define those function in the global workspace of our current R session, calling `pkgcheck()` again will automatically detect those checks and include them in our output:

```
#>
#> ── pkgstats 9.9␣
↪──────────────────────────────────────────────────────────────────────────────
#>
#> ✓ Package name is available
#> ✖ does not have a 'codemeta.json' file.
#> ✖ does not have a 'contributing' file.
#> ✓ uses 'roxygen2'.
#> ✓ 'DESCRIPTION' has a URL field.
#> ✓ 'DESCRIPTION' has a BugReports field.
#> ✖ Package has no HTML vignettes
#> ✖ These functions do not have examples: [pkgstats_from_archive].
#> ✓ Package has continuous integration checks.
#> ✖ Package coverage failed
#> ✖ Package name [pkgstats] does NOT start with 'aa'
#> ✖ R CMD check found 1 error.
#> ✓ R CMD check found no warnings.
#>
#>  Current status:
#> ✖ This package is not ready to be submitted.
```

Customised personal checks can be incorporated by defining them in a local package, loading that into the workspace, and passing the name of the package to the `extra_env` parameter.

3.2 New pkgcheck Checks (*for pkgcheck developers*)

New checks can be added to this package by creating new files in the `/R` directory prefixed with `pkgchk_`, and including the two functions described above (a check and an output function). The check name will then need to be included in the `order_checks()` function in the `R/summarise-checks.R` file, which determines the order of checks in the `summary` output. Checks which are not defined in this ordering, including any defined via `extra_env` parameters, appear *after* all of the standard checks, and prior to the `R CMD check` results which always appear last. This order may only be modified by editing the list in that function. The order of check results in the `print` method is also hard-coded, defined in the main `print.pkgcheck` method. As explicitly stated in that function, any new checks should also be included in the `print` method just after the first reference to `"misc_checks"`, via an additional line:

```
print_check_screen (x, "<name-of-new-check>", pkg_env)
```

The `print_check_screen()` function will then automatically activate the `print` method of any new checks. This line should be added even if a new check has no `print` method (as in the `starts_with_aa` example above), to provide an explicit record of all internally-defined miscellaneous checks.

Finally, any new checks also need to be included in tests. The test suites run on generic, mostly empty packages constructed with the `srr::srr_stats_pkg_skeleton()` function, as in the main `test-pkgcheck.R` test functions. Additional tests are also performed on the `pkgstats` tarball illustrated above. The default results of any new checks will be automatically tested by the existing test suite, but it is important to test all potential results. The `test-extra-checks.R` file is the main location for testing additional tests, with lines in that file demonstrating how the main results can be readily modified to reflect alternative outputs of check functions (such as `pkgchk_has_scrap` and `pkgchk_obsolete_pkg_deps`). The output functions defined as part of checks, including any new checks, do not need to be explicitly tested, as the entire output is tested via `testthat` snapshots. Snapshot results need to be updated to reflect any additional tests. Finally, the `test-list-checks.R` file tests the total number of internally-defined checks as `expect_length (ncks, ..)`. The number tested there also needs to be incremented by one for each new check.

4.11.2 NA

The following checks are currently implemented in the `pkgcheck` package. Several of these are by default only shown when they fail; absence from a resultant checklist may be interpreted to indicate successful checks.

1. pkgchk_fns_have_exs

Check whether all functions have examples.

2. pkgchk_fns_have_return_vals

Check that all functions document their return values.

The reflects a CRAN checks for all new submissions, to ensure that return values are documented for all functions. This check applies even to functions which are called for their side effects and return `NULL`.

3. pkgchk_has_citation

Check whether a package has a `inst/CITATION` file.

"CITATION" files are required for all rOpenSci packages, as documented in our "*Packaging Guide*. This does not check the contents of that file in any way.

4. pkgchk_has_codemeta

Check whether a package has a `codemeta.json` file.

"codemeta.json" files are recommended for all rOpenSci packages, as documented in our "*Packaging Guide*.

5. pkgchk_has_contrib_md

Check whether package has contributor guidelines in a 'contributing' file.

A "contributing" file is required for all rOpenSci packages, as documented in our "*Contributing Guide*.

6. pkgchk_license

Check whether the package license is acceptable.

Details of acceptable licenses are provided in the links in our *Packaging Guide*.

7. pkgchk_obsolete_pkg_deps

Check whether the package depends on any obsolete packages for which better alternative should be used.

The list of obsolete packages is given in our *Packaging Guide*. Some of these are truly obsolete, the use of which raises a red cross in the summary of checks; while others are only potentially obsolete, thus use of which merely raises a note in the detailed check output.

8. pkgchk_on_cran

Check whether a package is on CRAN or not.

This does not currently appear in any 'pkgcheck' output (that is, neither in summary or print methods), and is only used as part of `pkcchk_pkgname_available`.

9. pkgchk_pkgname_available

Check that package name is available and/or already on CRAN.

10. pkgchk_renv_activated

For packages which use 'renv', check that it is de-activated.

Although we do not generally recommend 'renv' for package developement, packages may use it. They must, however, ensure that `renv` is deactivated.

11. pkgchk_has_scrap

Check whether the package contains any useless files like `.DS_Store`.

Files currently considered "scrap" are:

1. ".DS_Store"
2. "Thumbs.db"
3. ".vscode"
4. ".o" files

12. pkgchk_unique_fn_names

Check whether all function names are unique.

Uses the database of function names from all CRAN packages associated with releases of the `pkgstats` package.

13. pkgchk_has_url

Check that a package 'DESCRIPTION' file lists a valid URL.

14. pkgchk_has_bugs

Check that a package 'DESCRIPTION' file lists a valid URL in the "BugReports" field.

15. pkgchk_uses_roxygen2

Check whether all documntation has been generated by 'roxygen2'

Use of 'roxygen2' to generate package documnetation is mandatory for rOpenSci packages, and is described in detail in our *Packaging Guide*.

16. pkgchk_has_vignette

Check whether the package contains at least one vignette.

17. pkgchk_left_assign

Check that left-assignment operators are used consistently throughout a package.

Left-assign operators recognised by R are: ("=", "<-", "<<-", ":="), as defined in the main grammar definition. This actually checks the following two conditions:

1. Check that any "global assignment operators" ("<<-") are only used in appropriate contexts; and
2. Check that all left-assignment operators are consistent, so either all use "=", or all use "<-", but do not mix these two symbols.

The `:=` operator is ignored in these checks.

18. pkgchk_srr

Check that 'srr' documentation for statistics pacakges is complete.

Procedures for preparing and submitting statistical packages are described in our "*Stats Dev Guide*". Statistical packages must use the 'srr' (software review roclets) package to document compliance with our statistical standards. This check uses the `srr::srr_stats_pre_submit()` function to confirm that compliance with all relevant standards has been documented.

19. pkgchk_ci_badges

Get all CI badges from a repository, and check that jobs currently pass.

This extracts badges directly from README files. If none are found and a repo is GitHub, then an additional function is called to check CI status directly from GitHub workflow files.

CHAPTER

FIVE

{PKGCHECK} GITHUB ACTION

This package contains a Github Actions script to automatically check R packages with rOpenSci's {pkgcheck} system which is run on all packages submitted for peer review. This action will implement a GitHub Action to ensure your package is ready to submit to rOpenSci's peer review system. The results are shown in the workflow output, and can also be posted in a new or updated GitHub Issue.

5.1 Usage

Like the GitHub actions functions from the `usethis` package, the `pkgcheck::use_github_action_pkcheck()` function will create a workflow file called `pkgcheck.yaml` in your local `.github/workflows` directory. The default workflow has the following relatively simple structure:

```yaml
name: pkgcheck

# This will cancel running jobs once a new run is triggered
concurrency:
  group: ${{ github.workflow }}-${{ github.head_ref }}
  cancel-in-progress: true

on:
  # Manually trigger the Action under Actions/pkgcheck
  workflow_dispatch:
  # Run on every push to main
  push:
    branches:
      - main

jobs:
  pkgcheck:
    runs-on: ubuntu-latest
    steps:
      - uses: ropensci-review-tools/pkgcheck-action@main
```

There are also several parameters which can be used to modify the workflow, as described in the following section.

5.1.1 Workflow parameters

The yaml workflow file which defines this action includes the following list of inputs:

```yaml
inputs:
  ref:
    description: "The ref to checkout and check. Set to empty string to skip checkout."
    default: "${{ github.ref }}"
    required: true
  post-to-issue:
    description: "Should the pkgcheck results be posted as an issue?"
    # If you use the 'pull_request' trigger and the PR is from outside the repo
    # (e.g. a fork), the job will fail due to permission issues
    # if this is set to 'true'. The default will prevent this.
    default: ${{ github.event_name != 'pull_request' }}
    required: true
  issue-title:
    description: "Name for the issue containing the pkgcheck results. Will be created or␣
↪updated."
    # This will create a new issue for every branch, set it to something fixed
    # to only create one issue that is updated via edits.
    default: "pkgcheck results - ${{ github.ref_name }}"
    required: true
  summary-only:
    description: "Only post the check summary to issue. Set to false to get the full␣
↪results in the issue."
    default: true
    required: true
  append-to-issue:
    description: "Should issue results be appended to existing issue, or posted in new␣
↪issues."
    default: true
    required: true
```

The easiest way to customise these inputs is with the pkgcheck::use_github_action_pkgcheck() function in R, the documentation of which includes the following example:

```
use_github_action_pkgcheck (inputs = list (`post-to-issue` = "false"))
```

That will produce a .github/workflows/pkgcheck.yaml file (or will update an existing file by setting the additional parameter, overwrite = TRUE) with the job: section changed from the default version shown above of:

```yaml
jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: ropensci-review-tools/pkgcheck-action@main
```

to the modified version of:

```yaml
jobs:
  check:
    runs-on: ubuntu-latest
    steps:
```

(continues on next page)

(continued from previous page)

```
    - uses: ropensci-review-tools/pkgcheck-action@main
      with:
        summary-only: false
```

That demonstrates that setting any of these parameters to non-default values by passing them as named list items to `pkgcheck::use_github_action_pkgcheck()` appends additional `yaml` lines to the workflow file through the `with:` statement. Workflows can thus be controlled either from R as shown above, or by directly editing the workflow file using `with:` statements. In R:

- Parameters must be passed as named items of a list named "inputs"
- Parameter names containing dashes must be specified within backticks.
- Parameter values must be specified in quotation marks.

In yaml scripts, parameter names and values should be specified exactly "as is", without quotation marks or backticks.

5.1.2 Posting {pgkcheck} results to a GitHub issue in your repository

The default workflow file posts the {pkgcheck} results to an issue in the repository in which it was run. This requires the workflow to have write access to your repo, which is automatically the case for events triggered within your repository such as pushes and pull requests from collaborators with write access.

If a pull request is opened from outside the repository such as from a fork, the default `github.token` will not have write access, and so will not be able to put results in an issue. This default behaviour protects your repository from malicious use of `pull_request` triggers.

:warning::warning: ***Never use the `pull_request_target` trigger as this will allow forks to run arbitrary code with access to your repos secrets***:warning::warning: For more information see here.

The first time this action is run, {pkgcheck} results will be created in a new issue of your repository. By default, each subsequent run will then append results to the same issue. The issue may be closed at any time, and results will still appear.

5.2 Versions

This action has no version tags, as you will always want to pass the newest {pgkcheck} available.

5.3 Contributors

All contributions to this project are gratefully acknowledged using the `allcontributors` package following the all-contributors specification. Contributions of any kind are welcome!

5.4 Functions

CHAPTER

SIX

ROREVIEWAPI

Plumber API to generate reports on package structure and function for the `@ropensci-review-bot`. The package is not intended for general use, and these documents are primarily intended for maintainers of this package, although they may serve as useful templates for similar endeavours. Please contact us if you have any questions.

Uses functionality provided by the `pkgcheck` and `pkgstats` packages. This suite of three packages requires a few system installs, two for `pkgstats` of `ctags` and GNU `global`. Procedures to install these libraries on various operating systems are described in the `pkgstats` package. This package itself also requires both the GitHub command-line-interface (cli), `gh` and `dos2unix`.

A local GitHub token also needs to be stored as an environment variable named `GITHUB_TOKEN`, and not `GITHUB_PAT` or anything else; the `gh` cli only recognises the former name.

The package also works by locally caching previously analysed packages, in a `pkgcheck` subdirectory of the location determined by

```
rappdirs::user_cache_dir()
```

You may manually erase the contents of this subdirectory at any time at no risk.

6.1 Dockerfile

The server associated with this package can be built by cloning this repository, and modifying the associated `Dockerfile` by inserting a GitHub token, and associated `git config` options.

6.2 Code of Conduct

Please note that this package is released with a Contributor Code of Conduct. By contributing to this project, you agree to abide by its terms.

6.3 Contributors

All contributions to this project are gratefully acknowledged using the `allcontributors` package following the all-contributors specification. Contributions of any kind are welcome!

6.3.1 Code

6.3.2 Issue Authors

6.3.3 Issue Contributors

6.4 Functions

6.4.1 check_cache

check_cache

Description

Check whether a package has been cached, and if so, whether commits have been added to the github repo since cached version.

Usage

```
check_cache(org, repo, cache_dir = fs::path_temp())
```

Arguments

Argument	Description
`org`	Github organization
`repo`	Github repository
`cache_dir`	Directory in which packages are cached

Value

FALSE If a package has previously been cached, and the github repo has not changed; otherwise TRUE.

Seealso

Other utils: *pkgrep_install_deps* , *stdout_stderr_cache* , *symbol_crs* , *symbol_tck*

Note

This function is not intended to be called directly, and is only exported to enable it to be used within the plumber API.

6.4.2 check_issue_template

Check template variables in GitHub issue

Description

Check template variables in GitHub issue

Usage

```
check_issue_template(orgrepo, issue_num)
```

Arguments

Argument	Description
orgrepo	GitHub organization and repo as single string separated by forward slash (org/repo).
issue_num	Number of issue from which to extract opening comment

Value

Comment as character string

Seealso

Other ropensci: *is_user_authorized* , *push_to_gh_pages* , *readme_has_peer_review_badge* , *srr_counts_summary* , *srr_counts* , *stats_badge*

6.4.3 collate_editor_check

Collate list of checks to single concatenated character string

Description

Collate list of checks to single concatenated character string

Usage

```
collate_editor_check(checks)
```

Arguments

Argument	Description
checks	Result of pkgcheck::pgkcheck function

Value

Single character

Seealso

Other main: *editor_check* , *serve_api*

Note

Exported only for access by plumber; not intended for general external usage.

6.4.4 dl_gh_repo

Download a GitHub repo to local cache

Description

Download a GitHub repo to local cache

Usage

```
dl_gh_repo(u, branch = NULL)
```

Arguments

Argument	Description
u	URL of GitHub repository
branch	Checkout specified (non-default) branch of repo.

Value

Path to locally cached '.zip' version of repository

Seealso

Other github: *get_branch_from_url* , *post_to_issue*

6.4.5 editor_check

Body of main 'editorcheck' response

Description

Body of main 'editorcheck' response

Usage

```
editor_check(repourl, repo, issue_id, post_to_issue = TRUE)
```

Arguments

Argument	Description
repourl	The URL for the repo being checked, potentially including full path to non-default branch.
repo	The 'context.repo' parameter defining the repository from which the command was invoked, passed in 'org/repo' format.
issue_id	The id (number) of the issue from which the command was invoked.
post_to_issu	Integer value > 0 will post results back to issue (via 'gh' cli); otherwise just return character string with result.

Value

If `!post_to_issue` , a markdown-formatted response body from static package checks, otherwise URL of the issue comment to which response body has been posted.

Seealso

Other main: *collate_editor_check* , *serve_api*

6.4.6 get_branch_from_url

Get branch from a GitHub URL if non-default branch specified there

Description

Get branch from a GitHub URL if non-default branch specified there

Usage

```
get_branch_from_url(repourl)
```

Arguments

Argument	Description
repourl	Potentially with "/tree/branch_name" appended

Value

Branch as single string.

Seealso

Other github: *dl_gh_repo* , *post_to_issue*

6.4.7 is_user_authorized

Check whether a user, identified from GitHub API token, is authorized to call endpoints.

Description

This function is used only in the plumber endpoints, to prevent them being called by unauthorized users.

Usage

```
is_user_authorized(secret = NULL)
```

Arguments

Argument	Description
secret	Environment variable PKGCHECK_TOKEN sent from bot.

Value

Logical value indicating whether or not a user is authorized.

Seealso

Other ropensci: *check_issue_template* , *push_to_gh_pages* , *readme_has_peer_review_badge* , *srr_counts_summary* , *srr_counts* , *stats_badge*

6.4.8 pkgrep_install_deps

Install all system and package dependencies of an R package

Description

Install all system and package dependencies of an R package

Usage

```
pkgrep_install_deps(path, repo, issue_id)
```

Arguments

Argument	Description
path	Path to local file or directory
repo	The 'context.repo' parameter defining the repository from which the command was invoked, passed in 'org/repo' format.
issue_id	The id (number) of the issue from which the command was invoked.

Value

Hopefully a character vector of length zero, otherwise a message detailing any R packages unable to be installed.

Seealso

Other utils: *check_cache* , *stdout_stderr_cache* , *symbol_crs* , *symbol_tck*

6.4.9 post_to_issue

Post review checks to GitHub issue

Description

Post review checks to GitHub issue

Usage

```
post_to_issue(cmt, repo, issue_id)
```

Arguments

Argument	Description
cmt	Single character string with comment to post.
repo	The repository to post to (obtained directly from bot).
issue_id	The number of the issue to post to.

Value

URL of the comment within the nominated issue

Seealso

Other github: *dl_gh_repo* , *get_branch_from_url*

6.4.10 push_to_gh_pages

Push static `html` files to `gh-pages` branch of this repo to serve via GitHub pages.

Description

Push static `html` files to `gh-pages` branch of this repo to serve via GitHub pages.

Usage

```
push_to_gh_pages(check)
```

Arguments

Argument	Description
`check`	Return result of *editor_check* function.

Value

Vector of two paths containing URLs of the `srr` and `network` files.

Seealso

Other ropensci: *check_issue_template* , *is_user_authorized* , *readme_has_peer_review_badge* , *srr_counts_summary* , *srr_counts* , *stats_badge*

6.4.11 readme_badge

Check whether README.md features an rOpenSci software-review badge

Description

Check whether README.md features an rOpenSci software-review badge

Usage

```
readme_badge(repourl, repo = NULL, issue_id = NULL, post_to_issue = TRUE)
```

Arguments

Argument	Description
`repourl`	The URL for the repo being checked, potentially including full path to non-default branch.
`repo`	The 'context.repo' parameter defining the repository from which the command was invoked, passed in 'org/repo' format.
`issue_id`	The id (number) of the issue from which the command was invoked.
`post_to_issu`	Integer value > 0 will post results back to issue (via 'gh' cli); otherwise just return character string with result.

Value

A string, empty if the badge was found.

6.4.12 readme_has_peer_review_badge

Check whether 'README.md' has a "peer reviewed" badge

Description

Check whether 'README.md' has a "peer reviewed" badge

Usage

```
readme_has_peer_review_badge(path = getwd(), issue_id = NULL)
```

Arguments

Argument	Description
path	Local path to package directory.
issue_id	The id (number) of the issue from which the command was invoked.

Value

A string, empty if the badge was found.

Seealso

Other ropensci: *check_issue_template* , *is_user_authorized* , *push_to_gh_pages* , *srr_counts_summary* , *srr_counts* , *stats_badge*

6.4.13 roreviewapi-package

roreviewapi: Plumber API to report package structure and function

Description

Plumber API to report package structure and function.

Seealso

Useful links:

- https://docs.ropensci.org/roreviewapi
- https://github.com/ropensci-review-tools/roreviewapi
- Report bugs at https://github.com/ropensci-review-tools/roreviewapi/issues

Author

Maintainer : Mark Padgham mark.padgham@email.com (ORCID)

6.4.14 serve_api

serve plumber API to report on packages

Description

The API exposes the single POST points of `report` to download software from the given URL and return a textual analysis of its structure and functionality.

Usage

```
serve_api(port = 8000L, cache_dir = NULL, os = "", os_release = "")
```

Arguments

Argument	Description
port	Port for API to be exposed on
cache_dir	Directory where previously downloaded repositories are cached
os	Name of operating system, passed to remotes package to install system dependencies.
os_release	Name of operating system release, passed to remotes package to install system dependencies.

Value

Nothing; calling this starts a blocking process.

Seealso

Other main: *collate_editor_check* , *editor_check*

6.4.15 srr_counts_summary

Summarise counts of 'srr' standards from full 'srr' report

Description

Summarise counts of 'srr' standards from full 'srr' report

Usage

```
srr_counts_summary(srr_rep)
```

Arguments

Argument	Description
srr_rep	An 'srr' report generated by the srr::srr_report() function.

Value

Character vector with markdown-formatted summary summary of numbers of standards complied with.

Seealso

Other ropensci: *check_issue_template* , *is_user_authorized* , *push_to_gh_pages* , *readme_has_peer_review_badge* , *srr_counts* , *stats_badge*

6.4.16 srr_counts

Count number of 'srr' statistical standards complied with, and confirm whether than represents > 50% of all applicable standards.

Description

Count number of 'srr' statistical standards complied with, and confirm whether than represents > 50% of all applicable standards.

Usage

```
srr_counts(repourl, repo, issue_id, post_to_issue = TRUE)
```

Arguments

Argument	Description
repourl	The URL for the repo being checked, potentially including full path to non-default branch.
repo	The 'context.repo' parameter defining the repository from which the command was invoked, passed in 'org/repo' format.
issue_id	The id (number) of the issue from which the command was invoked.
post_to_issu	Integer value > 0 will post results back to issue (via 'gh' cli); otherwise just return character string with result.

Value

Vector of three numbers:

- Number of standards complied with
- Total number of applicable standards
- Number complied with as proportion of total

Seealso

Other ropensci: *check_issue_template* , *is_user_authorized* , *push_to_gh_pages* , *readme_has_peer_review_badge* , *srr_counts_summary* , *stats_badge*

6.4.17 stats_badge

Get stats badge grade and standards version for a submission

Description

Get stats badge grade and standards version for a submission

Usage

```
stats_badge(repo = "ropensci/software-review", issue_num = 258)
```

Arguments

Argument	Description
repo	The submission repo
issue_num	GitHub issue number of submission

Value

A single character containing the label used directly for the issue badge

Seealso

Other ropensci: *check_issue_template* , *is_user_authorized* , *push_to_gh_pages* , *readme_has_peer_review_badge* , *srr_counts_summary* , *srr_counts*

6.4.18 stdout_stderr_cache

Set up stdout & stderr cache files for r_bg process

Description

Set up stdout & stderr cache files for r_bg process

Usage

```
stdout_stderr_cache(repourl)
```

Arguments

Argument	Description
repourl	The URL of the repo being checked

Value

Vector of two strings holding respective local paths to stdout and stderr files for r_bg process controlling the main *editor_check* function.

Seealso

Other utils: *check_cache* , *pkgrep_install_deps* , *symbol_crs* , *symbol_tck*

Note

These files are needed for the callr `r_bg` process which controls the main *editor_check* call. See https://github.com/r-lib/callr/issues/204 . The `stdout` and `stderr` pipes from the process are stored in the cache directory so they can be inspected via their own distinct endpoint calls.

6.4.19 symbol_crs

Cross symbol, exported for direct use in plumber API

Description

Cross symbol, exported for direct use in plumber API

Usage

```
symbol_crs()
```

Seealso

Other utils: *check_cache* , *pkgrep_install_deps* , *stdout_stderr_cache* , *symbol_tck*

6.4.20 symbol_tck

Tick symbol, exported for direct use in plumber API

Description

Tick symbol, exported for direct use in plumber API

Usage

```
symbol_tck()
```

Seealso

Other utils: *check_cache* , *pkgrep_install_deps* , *stdout_stderr_cache* , *symbol_crs*

6.5 Vignettes

The main API endpoint for the Digital Ocean server is the editorcheck, called by the ropensci-review-bot on every submission, and also manually (re-)triggered by the command @ropensci-review-bot check package. This vignette describes procedures which may be used to debug any instances in which package checks fail.

Most failures at the server end will deliver HTTP error codes of "500" like this example. This code *may* indicate that the server is currently unavailable, which will arise once per week during rebuild and redeploy processes, scheduled every Monday at 23:59 UTC. If a submission happens to be roughly at this time, the recommended procedure is to wait at least an hour before manually trying @ropensci-review-bot check package. Other 500 codes should be reported straight to maintainers of the check system, currently to @mpadge and @noamross as respective first and second points of contact. This vignette describes the procedures they would follow to debug any problems.

Debugging generally requires stepping through the code as called by the main editorcheck endpoint, and into the editor_check() function called by the endpoint as a background process.

6.5.1 Debugging Procedure

Check that the API is online

The following code will confirm that the API is online, by returning a single numeric value:

```
httr::GET ("http://<ip-address>:8000/mean") |>
    httr::content ()
```

Check log of recent requests

The log entries are illustrated in the endpoints vignette, and can be used to ensure that all variables were successfully delivered in the request. Any missing or malformed variables most likely indicate problems with the submission template. These should be caught by the initial call make by the editorcheck function to the check_issue_template() function. Potential issues can be debugged by calling that function locally:

```
roreviewapi::check_issue_template (orgrepo = "ropensci/software-review",
                                   issue_num = <issue_num>)
```

That should return an empty string with an additional attribute, "proceed_with_checks", which should be TRUE. Any other return value indicates an issue with the submission template, for which the return value should contain text describing the problem.

Check installation of system dependencies

This and the following checks are components of the `editor_check()` function called by the main endpoint as a background process, the first step of which is to identify and install system dependencies. This step is error prone, as is also the case on all CRAN machines. A final step of the check is to identify any packages which were unable to be installed, and to post a list of these directly in the submission issue. Such instances should generally be temporary, and fixed by forthcoming CRAN updates. These happen because of temporary breakages in one package which lead to other packages becoming unable to be installed from CRAN, with these temporary breakages in turn generally arising because of changes to external (non-R) system libraries. If any notified inabilities to install packages adversely affect final check results, debugging may require manually running checks on the Digital Ocean server, as described in the final section below.

Check system output and error logs

The system output and error logs from checks for a specified package are accessible from the `stdlogs` endpoint, which requires the single parameter of the `repourl` of the repository being checked. Logs are kept for the latest git head, and are accessible if and only if the latest GitHub commit matches the points at which the logs were generated. In those cases, the return value may offer useful diagnostic insights into potential problems either with submitted packages, or the check system itself.

6.5.2 Manually running checks

If all else fails, checks can be manually run directly from the Digital Ocean server, and sent straight to the relevant issue. The following procedure describe how, generally following the main `editor_check()` function.

1. Enter the `roreviewapi` Docker image via `docker run -it --rm roreviewapi /bin/bash` and start R.
2. Set `repourl <- <url>` and run `path <- roreviewapi::dl_gh_repo (repourl)` to download a clone of the repository.
3. Type `os <- "ubuntu"` and `os_release <- "20.04"`.
4. Run `p <- roreviewapi::pkgrep_install_deps (path, os, os_release)`. The value, `p`, will list any packages which were unable to be installed. These will then need to be manually installed, generally through finding the remote/dev URLs for the packages, and running `remotes::install_github()` or similar. Note that successful installation may only be possible in a particular order, and in the worst cases may be a process of trial and error.
5. Finally generate the main checks by running `checks <- pkgcheck::pkgcheck(path)`, during which diagnostic output will be dumped directly to the screen.
6. Convert checks to markdown format by running `out <- roreviewapi::collate_editor_check (checks)`.
7. Finally, post the checks to the desired issue with `out <- roreviewapi::post_to_issue (out, orgrepo, issue_num)`. Alternative values for `orgrepo` and `issue_num` can be used to first confirm that the checks look okay before posting them to `ropensci/software-review`.

These steps can be run in two code chunks, the first directly in the shell environment of the Digital Ocean droplet:

```
docker run -it --rm roreviewapi /bin/bash
R
```

And the second within the R environment:

```
repourl <- "https://github.com/org/repo" # replace with actual org/repo values
path <- roreviewapi::dl_gh_repo (repourl)
os <- "ubuntu"
os_release <- "20.04"
p <- roreviewapi::pkgrep_install_deps (path, os, os_release)
checks <- pkgcheck::pkgcheck(path)
out <- roreviewapi::collate_editor_check (checks)
orgrepo <- "ropensci/software-review" # or somewhere else for testing purposes
out <- roreviewapi::post_to_issue (out, orgrepo, issue_num)
```

6.5.3 API Endpoints

This vignette provides brief summaries of the endpoints of the `roreviewapi` package. These are encoded within the identical `R/plumber.R` and `inst/plumber.R` files. All endpoints respond only to `GET` calls.

---

1. editorcheck

This is the main endpoint called by the `ropensci-review-bot` in response to package submission. The call itself is configured as part of an external service call in `ropensci-org/buffy, which passes the parameters specified there of:

1. `repo` The GitHub repository from where the call originates, generally `ropensci/software-review`;
2. `issue_id` as the number of the issue in `repo` describing the software submission; and
3. `repourl` as specified in the submission template, and specifying the GitHub repository of the software being submitted, also in the format `<org>/<repo>`.

This endpoint implements the following steps:

1. Call `roreviewapi::check_issue_template()` to check the existence and format of HTML variables included within the submission template. This function returns an empty string if the template is okay; otherwise a descriptive error message. The return value also includes a binary attribute, `"proceed_with_checks"`, which is set to `FALSE` only if `repourl` is improperly specified. In this case the function returns immediately with a text string describing the error. Otherwise the string is carried through to the next step:
2. The `pkgcheck::pkgcheck()` function is started as a background process, dumping both `stdout` and `stderr` messages to specified logfiles (see `stdlogs` endpoint, below).
3. Any messages generated above are prepended to a return message that the package checks have started, that message delivered back to the bot, and ultimately dumped in the issue thread.

All messages, and the results of the `pkgcheck::pkgcheck()` process, are dumped to the specified `issue_id` in the specified `repo`.

---

2. editorcheck_contents

The `editorcheck_contents` endpoint implements the main check functionality of the `editorcheck` endpoint without dumping any results to the specified issue. It is primarily intended to aid debugging any issues arising within checks, through the use of the `stdlogs` endpoint described below. This endpoint accepts the single argument of `repourl` only.

---

3. mean

A simple `mean` endpoint can be used to confirm that the server is running. It accepts a single integer value of `n`, and returns the value of `mean(rnorm(n))`.

---

4. stats_badge

This endpoint is used by the bot to extract the stats badge from those issues which have one, in the form `"6\ approved-bronze-v0.0.1"`. This is used in turn by the bot to respond to `mint` commands used to change badge grades.

---

5. log

The `log` endpoint accepts a single parameter, `n`, specifying the number of latest log entries to retrieve. An example of the log entry for this submission follows:

```
#> [1] "INFO [2021-10-07 16:48:14] 3.236.83.25 \"Faraday v1.7.1\" <ip>:8000 GET /
↪editorcheck ?bot_name=ropensci-review-bot&issue_author=ewallace&issue_id=470&
↪repo=ropensci%2Fsoftware-review&repourl=https%3A%2F%2Fgithub.com%2Fewallace%2Ftidyqpcr&
↪sender=ewallace 200 1.964"
```

Each entry contains the following information:

1. Date and time at which call was made;
2. IP address and machine from which call was sent;
3. Method used to send call;
4. IP address to which call was delivered (always the address hosting the `roreviewapi` instance);
5. `http` method for the call (always `GET` for all endpoints encoded here);
6. The endpoint called (one of the methods listed above);
7. The parameters submitted along with the call;
8. The HTTP status of the call (hopefully 200); and
9. The total duration of the call response.

The 7th item of parameters submitted along with the call is particularly useful for debugging purposes; and is specified in this line of the `R/api.R` file.

---

6. clear_cache

This endpoint can be used to clear the server's cache whenever desired or required. This cache is mainly used to store the results of calls to `pkgcheck::pkgcheck()`. The only effect of clearing the cache will be extra time taken to regenerate any calls which were previously cached.

---

7. stdlogs

This is the most important endpoint for debugging problems within the `pkgcheck` process itself. The endpoint accepts the single parameter of `repourl`, and will return the results of both `stdout` and `stderr` connections produced during `pkgcheck`. These checks are hashed with the latest git head, ensuring that the endpoint returns checks for the latest commit.

CHAPTER

SEVEN

SRR

"srr" stands for **S**oftware **R**eview **R**oclets, and is rOpenSci's package for extending documentation to include additional components specific to the software review process. The package currently facilitates documenting how statistical software complies with our collections of Statistical Software Standards. Before proceeding, the answer to an important question: **What is a "roclet"?**

- A roclet is an object used by the `roxygen2` package to convert `roxygen2`-style documentation lines into some desired form of output.

7.1 Why then?

This package currently serves to aid developers and reviewers of statistical software in aligning their software against our extensive lists of standards. In acknowledgement of Colin Gillespie's sentiments expressed in his keynote speech at the European R Users Meeting 2020:

> Standards are good Standards should be strict No-one reads standards

the `srr` package aims to integrate the task of aligning software with standards within the practice of coding itself, and to make standards adherence as painless as possible.

7.2 How?

The `roxygen2` package parses all documentation lines from all files in the `R/` directory of a package which begin with `#'`. Special tags beginning with `@`, such as `@param` or `@export`, may follow these symbols, and roclets define what is done with different kinds of tags. The `roxygen2` package includes roclets to process a number of tags; the `srr` package implements custom roclets to process several additional tags for use with rOpenSci's software review systems.

At present, the package only contains roclets and associated functions to help those developing and reviewing packages submitted to rOpenSci's system for Statistical Software Review. The functions are mostly intended to ease alignment and assessment of software against the standards detailed in the main project book (from here on referred to as the "SSR Book").

7.3 Installation

The easiest way to install this package is via the associated r-universe. As shown there, simply enable the universe with

```
options(repos = c(
    ropenscireviewtools = "https://ropensci-review-tools.r-universe.dev",
    CRAN = "https://cloud.r-project.org"))
```

And then install the usual way with,

```
install.packages("srr")
```

Alternatively, the package can be installed by running one of the following lines:

```
remotes::install_github ("ropensci-review-tools/srr")
pak::pkg_install ("ropensci-review-tools/srr")
```

and loaded for use with,

```
library (srr)
```

7.4 Overview

Both this README, and the main package vignette, describe the functionality of the package in the specific context of the statistical software review project. Both the roclet and all functions intended for use in this context are prefixed with srr_stats_. The remainder of this document is in two main sections. If you're developing a statistics package for submission to our peer review system, keep straight on reading. If you've been invited to review a package, you may skip the following section and just read the subsequent section. The general procedures for both developers and reviewers are described at length in the SSR book, with this README intended to provide supporting technical details.

Note that the srr package can be applied only within the working directory of a package. There are no package or path arguments to allow functions to be applied to packages anywhere other than in the current working directory.

7.5 For Package Developers

People intending to develop packages for submission to our system for peer reviewing statistical software will need to follow the following general steps. Note that, while the srr package has a few functions which developers may call directly to aid their submission process, most functionality of this package is implemented via custom roxygen2 "roclets". The third of the following steps describes how to link your package with srr in order to use these roclets.

1. Ensure that your package successfully passes all autotest tests, potentially including setting test = FALSE flags to switch off any tests you consider not to be applicable to your package. For details, please see the package documentation for autotest.
2. Decide which of our in-scope categories of statistical software best describe your package. The function srr_stats_categories() provides a list of currently developed categories for which standards have been developed, along with links to the online standards for each category:

   ```
   srr_stats_categories ()$title
   ```

```
## [1] "General"
## [2] "Bayesian"
## [3] "EDA"
## [4] "Machine Learning"
## [5] "Regression and Supervised Learning"
## [6] "Spatial"
## [7] "Time Series"
## [8] "Dimensionality Reduction, Clustering, and Unsupervised Learning"
```

That function also returns links to the full descriptions of each category in the main project book. Any software within one or more of these categories may be considered for review.

3. Enable your package to use the `srr_stats` roclets by modifying the package's `DESCRIPTION` file so that the `Roxygen` line looks like this:

```
Roxygen: list(markdown = TRUE, roclets = c ("namespace", "rd", "srr::srr_stats_
↪roclet"))
```

That will load the "roclet" used by this package to process the documentation of statistical standards within your actual code. Note that you do not need to add, import, or depend upon the `srr` package anywhere else within the `DESCRIPTION` file or your actual package.

4. Load the `srr` package and generate lists of standards within your package's `/R` folder by running, `srr_stats_roxygen(category = c("<my-category-1>", "<my-category-2>"))`. This will by default create a new file called by default `R/srr_stats_standards.R`, the first few lines of which will look like this:

```
## [1] "#' srr_stats"
## [2] "#'"
## [3] "#' All of the following standards initially have `@srrstatsTODO` tags."
## [4] "#' These may be moved at any time to any other locations in your code."
## [5] "#' Once addressed, please modify the tag from `@srrstatsTODO` to␣
↪`@srrstats`,"
## [6] "#' or `@srrstatsNA`, ensuring that references to every one of the following"
```

The file will contain a list of all standards from your nominated categories. This file may be renamed, and the individual items moved to other locations in other files, but all nominated standards should remain in `roxygen2` blocks somewhere in your source code.

The `@srrstatsVerbose` line defines a variable which may be used to suppress output from the `srrstats` roclet when updating package documentation (by setting to `FALSE`). After that comes the list of standards, each of which is prefixed by a `roxygen2` tag, `@srrstatsTODO`. A package can only be submitted once all of these `TODO` items have been addressed via one of the options described in the following two items.

5. A standard may be addressed by moving the item in the `srr-stats-standards.R` file (or wherever you've chosen to list these within your own package) to one or more places in your code where these standards have been addressed. In doing so, the `roxygen2` tag should be changed from `@srrstatsTODO` to `@srrstats`, and the text which initially lists the actual standard should be changed to provide a brief description of how that standard has been met. Tags for one particular standard may be repeated in multiple places within your code, and we encourage locating multiple `@srrstats` tags which refer to a particular standard at all locations which directly address that standard.

6. Alternatively, any standards which you consider not applicable to your software may remain listed in the templated section of the main `srr-stats-standards.R` document (or any alternative location), with their tag changed from `@srrstatsTODO` to `@srrstatsNA`, and the description of the standard removed and replaced by an explanation of why you consider that standard not to be applicable to your software. These `@srrstatsNA` tags should be collected together within a single `roxygen2` block with a title of `NA_standards`, as provided in

the initial template generated by the srr_stats_roxygen() function. Any non-applicable standards can then just be moved into this block, with their @srrstatsTODO tags changed to @srrstatsNA

7. Each time you run devtools::document() or the equivalent roxygen2::roxygenise(), the roclet will scan your package's documentation for the state of standards, and will generate a summary of the result on your screen.

To help developers understand how to use these roclets, this package includes a function, srr_stats_pkg_skeleton(), which will generate a skeleton of a package with several roxygen2 tags inserted throughout the code. This function returns the directory where the skeleton package has been created, so running the following two lines will illustrate the roclets in action:

```
d <- srr_stats_pkg_skeleton ()
roxygen2::roxygenise (d)
```

Note that the skeleton package also includes C++ code in a src/ directory, so will be compiled the first time your run roxygensise(). Running a second time will generate cleaner output from the srr_stats roclets only. The tags included in roxygen2 blocks in this skeleton package may be modified, moved, copied, and changed in any way you like to help you understand how the roclets work. Simply play around with the roxygen2 lines and run roxygensise() each time to see the effect. Individual standards may be moved to, and addressed in, any location including the directories R/, src/, or tests/, and well as in .Rmd documentation files such as README.Rmd or package vignettes. The srr_stats roclet associated with this package is able to parse the various @srrstats tags in all of these locations.

7.5.1 Places where standards can NOT be inserted

While the srr package enables standards compliance to be documented through inserting @srrstats tags in as many locations as possible, in order to ensure compliance is documented as close as possible to the point within the code where each standard is addressed, it is not possible to insert roxygen2 tags in every type of file. In general, standards may be inserted in any .R or .Rmd file, and most types of files in src or inst/include directories, as long as they are used with a package able to convert documentation to a corresponding R file (such as Rcpp's generation of RcppExports.R files which include the C++ documentation).

Tags may generally not be placed in any other kinds of files, including .md files such as CONTRIBUTING.md, or other files without extensions such as DESCRIPTION, NAMESPACE, or NEWS. Standards which are best addressed in such files must be placed in some other generic location (such as R/srr-standards.R), with a cross-reference to the file in which they are actually addressed.

7.6 Code of Conduct

Please note that this package is released with a Contributor Code of Conduct. By contributing to this project, you agree to abide by its terms.

7.7 Contributors

All contributions to this project are gratefully acknowledged using the allcontributors package following the all-contributors specification. Contributions of any kind are welcome!

7.7.1 Code

7.7.2 Issue Authors

7.7.3 Issue Contributors

7.8 Functions

7.8.1 srr_report

Generate report from ssr tags.

Description

Generate report from ssr tags.

Usage

```
srr_report(path = ".", branch = "", view = TRUE)
```

Arguments

Argument	Description
path	Path to package for which report is to be generated
brancl	By default a report will be generated from the current branch as set on the local git repository; this parameter can be used to specify any alternative branch.
view	If TRUE (default), a html-formatted version of the report is opened in default system browser. If FALSE , the return object includes the name of a html -rendered version of the report in an attribute named 'file'.

Value

(invisibly) Markdown-formatted lines used to generate the final html document.

Examples

```
path <- srr_stats_pkg_skeleton ()
srr_report (path)
```

7.8.2 srr_stats_categories

Get details of current statistical software categories

Description

List all currently available categories and associated URLs to full category descriptions.

Usage

```
srr_stats_categories()
```

Value

A `data.frame` with 3 columns of "category" (the categories to be submitted to *srr_stats_checklist*), "title" (the full title), and "url".

Seealso

Other helper: *srr_stats_checklist* , *srr_stats_checklist_check* , *srr_stats_pkg_skeleton* , *srr_stats_pre_submit*

Examples

```
srr_stats_categories ()
```

7.8.3 srr_stats_checklist_check

Check a completed standards checklist

Description

Correct any potential formatting issues in a completed standards checklist

Usage

```
srr_stats_checklist_check(file)
```

Arguments

Argument	Description
`file`	Name of local file containing a completed checklist. Must be a markdown document in `.md` format, not `.Rmd` or anything else.

Seealso

Other helper: *srr_stats_categories* , *srr_stats_checklist* , *srr_stats_pkg_skeleton* , *srr_stats_pre_submit*

Examples

```
f <- tempfile (fileext = ".md")
srr_stats_checklist (category = "regression", filename = f)
chk <- srr_stats_checklist_check (f)
```

7.8.4 srr_stats_checklist

Download checklists of statistical software standards

Description

Obtain rOpenSci standards for statistical software, along with one or more category-specific standards, as a checklist, and store the result in the local clipboard ready to paste.

Usage

```
srr_stats_checklist(category = NULL, filename = NULL)
```

Arguments

Argument	Description
`categ`	One of the names of files given in the directory contents of https://github.com/ropensci/statistical-software-review-book/tree/main/standards , each of which is ultimately formatted into a sub-section of the standards.
`filen`	Optional name of local file to save markdown-formatted checklist. A suffix of `.md` will be automatically appended.

Value

A character vector containing a markdown-style checklist of general standards along with standards for any additional categories.

Seealso

Other helper: *srr_stats_categories* , *srr_stats_checklist_check* , *srr_stats_pkg_skeleton* , *srr_stats_pre_submit*

Examples

```
x <- srr_stats_checklist (category = "regression")
# or write to specified file:
f <- tempfile (fileext = ".md")
x <- srr_stats_checklist (category = "regression", filename = f)
```

7.8.5 srr_stats_pkg_skeleton

Make skeleton package to test roclet system

Description

Make a dummy package skeleton including 'srr' roxygen2 tags which can be used to try out the functionality of this package. Running the example lines below which activate the 'srr' roclets, and show you what the output of those roclets looks like. Feel free to examine the effect of modifying any of the @srrstats tags within the code as identified by running those lines.

Usage

```
srr_stats_pkg_skeleton(base_dir = tempdir(), pkg_name = "demo")
```

Arguments

Argument	Description
base_dir	The base directory where the package should be constructed.
pkg_name	The name of the package. The final location of this package will be in file.path(base_dir, pkg_name) .

Value

The path to the directory holding the newly created package

Seealso

Other helper: *srr_stats_categories* , *srr_stats_checklist* , *srr_stats_checklist_check* , *srr_stats_pre_submit*

Examples

```
d <- srr_stats_pkg_skeleton (pkg_name = "mystatspkg")
# (capture.output of initial compliation messages)
x <- utils::capture.output (roxygen2::roxygenise (d), type = "output")
```

7.8.6 srr_stats_pre_submit

Perform pre-submission checks

Description

Check that all standards are present in code, and listed either as '@srrstats' or '@srrstatsNA'

Usage

```
srr_stats_pre_submit(path = ".", quiet = FALSE)
```

Arguments

Argument	Description
path	Path to local repository to check
quiet	If 'FALSE', display information on status of package on screen.

Value

(Invisibly) List of any standards missing from code

Seealso

Other helper: *srr_stats_categories* , *srr_stats_checklist* , *srr_stats_checklist_check* , *srr_stats_pkg_skeleton*

Examples

```
d <- srr_stats_pkg_skeleton ()
# The skeleton has 'TODO' standards, and also has only a few from the full
# list expected for the categories specified there.
srr_stats_pre_submit (d)
```

7.8.7 srr_stats_roclet

srr_stats_roclet

Description

Get values of all `srrstats` tags in function documentation

Usage

```
srr_stats_roclet()
```

Details

Note that this function should never need to be called directly. It only exists to enable "@srrstats" tags to be parsed from roxygen2 documentation.

Value

A roxygen2 roclet

Seealso

Other roxygen: *srr_stats_roxygen*

Examples

```
srr_stats_roclet ()
```

7.8.8 srr_stats_roxygen

Insert standards into code in roxygen2 format

Description

Obtain rOpenSci standards for statistical software, along with one or more category-specific standards, as a checklist, convert to project-specific roxygen2 format, and save in nominated file.

Usage

```
srr_stats_roxygen(
  category = NULL,
  filename = "srr-stats-standards.R",
  overwrite = FALSE
)
```

Arguments

Argument	Description
categc	One of the names of files given in the directory contents of https://github.com/ropensci/statistical-software-review-book/tree/main/standards , each of which is ultimately formatted into a sub-section of the standards.
filena	Name of 'R' source file in which to write roxygen2 -formatted lists of standards.
overwı	If FALSE (default) and filename already exists, a dialog will ask whether file should be overwritten.

Value

Nothing

Seealso

Other roxygen: *srr_stats_roclet*

Examples

```
path <- srr_stats_pkg_skeleton ()
# contains a few standards; insert all with:
f <- file.path (path, "R", "srr-stats-standards.R")
file.exists (f)
length (readLines (f)) # only 14 lines
srr_stats_roxygen (
category = "regression",
file = f,
```

(continues on next page)

(continued from previous page)

```
overwrite = TRUE
)
length (readLines (f)) # now much longer
```

7.8.9 srr-package

srr: 'rOpenSci' Review Roclets

Description

Companion package to 'rOpenSci' statistical software review project.

Seealso

Useful links:

- https://docs.ropensci.org/srr/
- https://github.com/ropensci-review-tools/srr
- Report bugs at https://github.com/ropensci-review-tools/srr/issues

Author

Maintainer : Mark Padgham mark@ropensci.org (ORCID)

7.9 Vignettes

7.9.1 Preparing Statistical Software with the srr package

`srr` stands for **S**oftware **R**eview **R**oclets, and is an R package containing roclets for general use in helping those developing and reviewing packages submitted to rOpenSci. At present, the `srr` package only contains roclets and associated functions to help those developing and reviewing packages submitted to rOpenSci's system for Statistical Software Review. This vignette demonstrates how developers are intended to use this package to document the alignment of their software with rOpenSci's standards for statistical software.

The main functions of the package are constructed as `roxygen2` "roclets", meaning that these functions are called each time package documentation is updated by calling `devtools::document()` or equivalently `roxygen2::roxygenise()`. The former is just a wrapper around the latter, to enable documentation to updated by a single call from within the `devtools` package. From here on, this stage of updating documentation and triggering the `srr` roclets will be referred to as "running `roxygenise`."

1. The package skeleton

The srr_stats_pkg_skeleton() function included with this package differs from other package skeleton functions. Rather than providing a skeleton from which you can construct your own package, the srr_stats_pkg_skeleton() function generates a skeleton to help developers understand this package works. This skeleton of a package is intended to be modified to help you understand which kinds of modifications are allowed, and which generate errors. The package is by default called "demo", is constructed in R's tempdir(), and looks like this:

```
library (srr)
d <- srr_stats_pkg_skeleton (pkg_name = "package")
fs::dir_tree (d)
```

```
## /tmp/RtmpuwzVqt/package
## ├── DESCRIPTION
## ├── NAMESPACE
## ├── R
## │   ├── RcppExports.R
## │   ├── package-package.R
## │   ├── srr-stats-standards.R
## │   └── test.R
## ├── README.Rmd
## ├── src
## │   ├── RcppExports.cpp
## │   └── cpptest.cpp
## └── tests
##     ├── testthat
##     │   └── test-a.R
##     └── testthat.R
```

The files listed there mostly exist to illustrate how standards can be included within code. The format of standards can be seen by examining any of those files. For example, the test.R file looks like this:

```
readLines (file.path (d, "R", "test.R"))
```

```
##  [1] "#' test_fn"
##  [2] "#'"
##  [3] "#' A test funtion"
##  [4] "#'"
##  [5] "#' @srrstats {G1.1, G1.2, G1.3} with some text"
##  [6] "#' @srrstats Text can appear before standards {G2.0, G2.1}"
##  [7] "#' @srrstatsTODO {RE1.1} standards which are still to be"
##  [8] "#' addressed are tagged 'srrstatsTODO'"
##  [9] "#'"
## [10] "#' @export"
## [11] "test_fn <- function() {"
## [12] "  message(\"This function does nothing\")"
## [13] "}"
```

That file illustrates some of the roxygen2 tags defined by this package, and described in detail below. These tags are parsed whenever package documentation is updated with roxygenise(), which will produce output like the following:

```
roxygen2::roxygenise (d)
```

```
## Setting `RoxygenNote` to "7.2.3"
##  Loading package
## Writing 'NAMESPACE'
##␣
↪──────────────────────────────────────────────────────────────────────────────
↪rOpenSci Statistical Software Standards␣
↪──────────────────────────────────────────────────────────────────────────────
##
##
##
## ── @srrstats standards (8 / 12):
##
##   * [G1.1, G1.2, G1.3, G2.0, G2.1] in function 'test_fn()' on line#11 of file [R/test.
↪R]
##   * [RE2.2] on line#2 of file [tests/testthat/test-a.R]
##   * [G2.3] in function 'test()' on line#6 of file [src/cpptest.cpp]
##   * [G1.4] on line#17 of file [./README.Rmd]
##
##
##
## ── @srrstatsNA standards (1 / 12):
##
##   * [RE3.3] on line#5 of file [R/srr-stats-standards.R]
##
##
##
## ── @srrstatsTODO standards (3 / 12):
##
##   * [RE4.4] on line#14 of file [R/srr-stats-standards.R]
##   * [RE1.1] on line#11 of file [R/test.R]
##   * [G1.5] on line#17 of file [./README.Rmd]
##
##␣
↪──────────────────────────────────────────────────────────────────────────────
##
## Writing 'package-package.Rd'
## Writing 'test_fn.Rd'
## Writing 'NAMESPACE'
```

The "roclets" contained within this package parse any instances of the package-specified tags described below, and summarise the output by listing all locations of each kind of tag. Locations are given as file and line numbers, and where appropriate, the names of associated functions. We recommend that developers familiarise themselves with the system by simply modifying any `roxygen2` block in any of the files of this package skeleton, and running `roxygenise()` to see what happens.

2. Enabling srr roclets for a package

The "roclets" can be enabled for a package by modifying the DESCRIPTION file so that the Roxygen line looks like this:

```
Roxygen: list (markdown = TRUE, roclets = c ("namespace", "rd", "srr::srr_stats_roclet"))
```

That will load the "roclets" used by this package to process standards as documented within your actual code. Note that you do not need to add, import, or depend upon the srr package anywhere else within the DESCRIPTION file. See the DESCRIPTION file of the package skeleton for an example.

3. roxygen2 tags

The srr packages recognises and uses the following three tags, each of which may be inserted into roxygen2 documentation blocks anywhere in your code. The tags are:

1. @srrstats tags to indicate standards which have been addressed. These should be placed at the locations within your code where specific standards have been addressed, and should include a brief textual description of how the code at that location addresses the nominated standard.
2. @srrstatsTODO tags to indicate standards which have not yet been addressed. The srr_stats_roxygen() function described below will place all standards for your nominated categories in an initial file in the /R directory of your project. Each of these will have a tag of @srrstatsTODO. As you address each standard, you'll need to move it from that initial location to that point in your code where that standard is addressed, and change the tag from @srrstatsTODO to @srrstats. It may help to break the initial, commonly very long, single list of standards into smaller groups, and to move each of these into the approximate location within your code where these are likely to be addressed. For example, all standards which pertain to tests can be moved into the /tests (or /tests/testthat) directory.
3. @srrstatsNA tags to indicate standards which you deem not applicable to your package. These need to be grouped together in a single block with a title of NA_standards. Such a block is both included in the package skeleton, and also in the output of srr_stats_roxygen() function. The numbers of any non-applicable standards can then be moved to this block, with a note explaining why these standards have been deemed not to apply to your package.

The output illustrated above shows how the srr package groups these three tags together, first collecting the output of standards which have been addressed (via @srrstats tags), then showing output of non-applicable standards (via @srrstatsNA tags), and finally standards which are still to be addressed (via @srrstatsTODO tags).

3.1 Format of srr documentation

The tags shown above from the skeleton package function test.R indicates the expected format of standards within roxygen2 documentation blocks. The package is designed to error on attempting to run roxygenise() with any inappropriately formatted entries, and so should provide informative messages to help rectify any problems. srr documentation must adhere to the following formatting requirements:

1. The tags must appear immediately after the roxygen2-style comment symbols. This will almost always mean `#' @srrstats` (but see details below, under *Locations of srrstats documentation*).
2. The standards number(s) must be enclosed within curly braces, such as `#' @srrstats {G1.0}`. Multiple standards may be associated with a single tag by including them within the same single pair of curly braces, and separating each by a comma, such as `#' @srrstats {G1.0, G1.1}`.
3. Explanatory text may be placed anywhere before or after curly braces enclosing standards, such that `#' @srrstats some text {G1.0}` is entirely equivalent to `#' @srrstats {G1.0} some text`.

4. Only the first pair of curly braces is parsed to determine standards numbers; any subsequent curly braces within one expression will be ignored. (Where an expression is everything that comes after one roxygen2 tag, and extends until the start of the next tag.) The following will accordingly only refer to G1.0, and not G1.1:

```
#' @srrstats {G1.0} as well as {G1.1}.
```

   The appropriate way to refer to multiple tags is to include them in the one set of curly braces, as shown above, or to use separate tags, like this:

```
#' @srrstats {G1.0} as well as
#' @srrstats {G1.1}
```

5. Any standard may occur arbitrarily many times in any file(s) within a package. The only requirement is that any one standard should only be associated with one single kind of tag; thus should only be `@srrstats` (a standard which has been addressed), *or* `@srrstatsNA` (a standard which is not applicable), *or* `@srrstatsTODO` (a standard which has not yet been addressed).

In almost all cases, all tags for a package will be generated by the initial call to `srr_stats_roxygen()`, and should simply be moved to appropriate locations within a package's code without modifying the format.

3.2 Locations of `srrstats` documentation

`@srrstats` tags and accompanying text can be placed almost anywhere within a package, especially in any file in the main `/R`, `/tests`, or `src/` directories. Within the `/R` directory, tags should be placed only in `roxygen2` blocks. These tags, and all associated text, will be ignored by the roclets used by `roxygen2` to generate package documentation, and will only appear on screen output like that shown above, and generated when running `roxygenise()`. If tags need to be inserted where there is no `roxygen2` block, then a new block will need to be created, the minimal requirement for which is that it then have an additional `#' @noRd` tag to suppress generation of documentation (`.Rd`) files. If that block is associated with a function, the following two lines will suffice:

```
#' @srrstats G1.0 This standard belongs here
#' @noRd
myfunction <- function (...) {
    # ...
}
```

If that block is not associated with a function, the documentation can be followed by a `NULL`, like this:

```
#' @srrstats G1.0 This standard belongs here
#' @noRd
NULL
```

Unlike `roxygen2`, which only processes blocks from within the main `R/` directory, the `srr` package process blocks from within other directories too. As these blocks will never be passed through to the main `roxygenise()` function, they need neither `#' @noRd` tags, nor `NULL` definitions where none otherwise exist. An example is in the `tests/` directory of the package skeleton:

```
readLines (file.path (d, "tests", "testthat", "test-a.R"))
```

```
## [1] "#' @srrstats {RE2.2} is addressed here"
## [2] "test_that(\"dummy test\", {"
## [3] "    expect_true (TRUE)"
## [4] "})"
```

While `@srrstats` tags can also be placed in the `src/` directory, the package currently only parses doxygen-style blocks for code written in C or C++. (Note that `srr` is currently further restricted to C++ code compiled with Rcpp, but will soon be adapted to work with other C++ interfaces such as cpp11.) These blocks are converted by Rcpp into roxygen2 blocks in a file called `R/RcppExports.R`, and so need to include an additional `@noRd` tag to (optionally) suppress generation of `.Rd` documentation. The skeleton package again gives an example:

```
readLines (file.path (d, "src", "cpptest.cpp"))
```

```
##  [1] "#include <Rcpp.h>"
##  [2] ""
##  [3] "//' src_fn"
##  [4] "//'"
##  [5] "//' A test C++ function"
##  [6] "//' @srrstats {G2.3} in src directory"
##  [7] "//' @noRd"
##  [8] "// [[Rcpp::export]]"
##  [9] "int test () {"
## [10] "    return 1L; }"
```

3.3 Documenting standards for documentation

Many standards refer to general package documentation, particularly in a `README` document, or in package vignettes. All such documents are presumed to be written in `.Rmd` format, for which `@srrstats` tags must be included within distinct code chunks. Again, the skeleton package has an example as follows:

```
readLines (file.path (d, "README.Rmd"))
```

```
##  [1] "# package"
##  [2] ""
##  [3] "This is a skeleton of an [`srr` statistics](https://github.com/ropensci-review-
↪tools/srr)"
##  [4] "package, intended developers to tweak as they like, in order to understand "
##  [5] "how the package's roclets work."
##  [6] ""
##  [7] "This `README.Rmd` file is here to demonstrate how to embed `srr` roclet tags."
##  [8] "These tags need to be within dedicated *code chunks*, like the following:"
##  [9] ""
## [10] "```{r srr-tags, eval = FALSE, echo = FALSE}"
## [11] "#' roxygen_block_name"
## [12] "#'"
## [13] "#' (Add some text if you like)"
## [14] "#'"
## [15] "#' @srrstats {G1.4} Here is a reference to a standard"
## [16] "#' @srrstatsTODO {G1.5} And here is a reference to a standard yet to be␣
↪addressed"
## [17] "```"
## [18] ""
## [19] "Note the chunk contains only [`roxygen2`](https://roxygen2.r-lib.org) lines,"
## [20] "and nothing else at all. Please change the `eval` and `echo` parameters to"
## [21] "see what happens when you knit the document."
```

Those lines illustrate the expected form. `@srrstats` tags should be within a single block contained within a dedicated code chunk. `@srrstats` chunks within `.Rmd` files will generally use `echo = FALSE` to prevent them appearing in

the rendered documents. The roxygen2 lines do not need to be followed by a NULL (or any other non- roxygen2 statement), although if additional R code is necessary for any reason, you may also need to add eval = FALSE.

3.4 @srrstatsNA tags for non-applicable standards

While @srrstatsTODO and @srrstats tags may be placed anywhere within a package, @srrstatsNA tags used to denote non-applicable standards must be placed within a dedicated roxygen2 block with a title of NA_standards. As described above, both the package skeleton and the file produced by calling srr_stats_roxygen() include templates for this block. The following illustrates a minimal form:

```
#' NA_standards
#'
#' @srrstatsNA {S3.3} is not applicable
#' @noRd
NULL
```

An NA_standards block must end with NULL, rather than be associated with a function definition. There can be multiple NA_standards blocks in any location, enabling these standards to be moved to approximate locations where they might otherwise have been addressed. (For example, non-applicable standards referring to tests might all be grouped together in a single NA_standards block in the tests/ directory.)

4. The srr workflow

The first step for any developer intending to use this package on the way to a submission to rOpenSci's project for peer-reviewing statistical software is to generate the package skeleton described above, and to try any and all conceivable ways to modify locations, formats, and other properties of the roxygen2 tags defined in this package, in order to understand how these tags are used to generate the summary results when running roxygenise. Following that initial familiarisation, a typical workflow will involve the following general steps:

1. Automatically download and insert lists of general and category-specific standards in your package by running srr_stats_roxygen() (in the main package directory). This will by default generate a file in the R/ directory called srr-stats-standards.R, although any alternative name can also be passed to the function (or the file can be renamed after it has initially been created). Each group of standards in this file must always be terminated by NULL in order to be appropriately parsed by the roxygen2 package.
2. Change your package's DESCRIPTION file to use the srr roclets by adding roclets = "srr::srr_stats_roclet") to the Roxygen line, as demonstrated at the outset, as well as in the package skeleton.
3. Run roxygenise to confirm that the roclets work on your package. You should initially see only a single list of @srrstatsTODO standards. All documented standards should appear somewhere in the output. Any missing standards indicate documentation problems, such as missing terminal NULL values from standards blocks.
4. We recommend as a first step cutting-and-pasting standards to approximate locations within your package's code where you anticipate these standards being addressed. Multiple copies of any one standard may exist in multiple locations, so you may also repeat standards which you anticipate will be addressed in multiple locations. This should reduce a potentially very long initial list of standards down to several distinct groups of hopefully more manageable size.
5. Begin addressing the standards by:
    - Ensuring your code conforms;
    - Moving each standard to the one or more location(s) where you think your code most directly addresses them;

- Modifying the `@srrstatsTODO` tag to `@srrstats`
- Changing the initial text describing the standard itself to a brief description of how your code addresses that standard.

6. Standards which you deem not to be applicable to your package should be grouped together in a single `roxygen2` block with the title `NA_standards` (as described above, and as generated by the `srr_stats_roxygen()` function). This block must finish with a `NULL` statement.
7. Update your documentation as frequently as you like or need, and use the output of the roclets to inform and guide the process of converting all `@srrstatsTODO` tags into either `@srrstats` or `@srrstatsNA` tags.

Note that we do **not** recommend copying files from the package skeleton into your own package for you `srr` documentation. The following lines demonstrate what happens if you insert actual standards into the package skeleton:

```
srr_stats_roxygen (category = "regression", # for example
                   filename = file.path (d, "R", "srr-stats-standards.R"),
                   overwrite = TRUE)
```

```
## ✓ Downloaded general standards
## ✓ Downloaded regression standards
##  Roxygen2-formatted standards written to [srr-stats-standards.R]
```

```
roxygen2::roxygenise (d)
```

```
##  Loading package
## Error: Standards [G1.1, G1.2, G1.3, G2.0, G2.1, RE2.2, G2.3, G1.4] are listed with␣
↪both @srrstats and @srrstatsTODO tags.
## Please rectify to ensure these standards are only associated with one tag.
```

To ensure all standards are first inserted with `@srrstatsTODO` tags, and that there are no duplicates with other tags, please use only the `srr_stats_roxygen()` function.

CHAPTER

EIGHT

ROPENSCI SOFTWARE REVIEW DASHBOARD

This repository contains the source code for rOpenSci's *Software Review Dashboard*. This includes both a local R package named "dashboard", and a quarto directory which includes the source files for the website. This README is intended only for developers. Anybody solely interested in the dashboard should head straight to the website.

8.1 renv and local usage

The quarto website uses `renv` to manage package dependencies in the GitHub workflow. The environment will be automatically built the first time R is started in the root directory of this repository. This environment will nevertheless not include the package itself, and so the package needs to be manually installed using:

```
remotes::install_github ("ropensci-review-tools/dashboard")
```

The website can be locally previewed by running `quarto preview` in the `quarto` directory, using the locally-installed version of the "dashboard" package. This means that any updates to the package itself will only be rendered on the quarto website once those changes have been pushed and the package locally re-installed using the `install_github` command above.

8.2 Functions

8.2.1 add_editor_airtable_data

Add additional columns to 'editors' data from rOpenSci's airtable database.

Description

Add additional columns to 'editors' data from rOpenSci's airtable database.

Usage

```
add_editor_airtable_data(editors)
```

Arguments

Argument	Description
`editors`	The `data.frame` of editors returned as the "status" component from *editor_status* .

Value

A modified version of `editors` with additional columns.

8.2.2 dashboard-package

dashboard: What the Package Does (One Line, Title Case)

Description

What the package does (one paragraph).

Seealso

Useful links:

- https://github.com/ropensci-review-tools/dashboard

Author

Maintainer : First Last first.last@example.com

8.2.3 editor_status

Generate a summary report of current state of all rOpenSci editors

Description

Generate a summary report of current state of all rOpenSci editors

Usage

```
editor_status(quiet = FALSE)
```

Arguments

Argument	Description
quiet	If FALSE, display progress information on screen.

Value

A data.frame with one row per issue and some key statistics.

8.2.4 editor_vacation_status

Get current vacation status of all rOpenSci editors.

Description

Get current vacation status of all rOpenSci editors.

Usage

```
editor_vacation_status()
```

Value

A data.frame with one row per editor and information on current vacation status obtained from rOpenSci's airtable database.

8.2.5 review_history

Generate historical data on software reviews.

Description

This is a reduced version of the reviews_gh_data() function, which returns data only on dates of issue opening and closing, to be used to generate historical patterns.

Usage

```
review_history(quiet = FALSE)
```

Arguments

Argument	Description
quiet	If FALSE, display progress information on screen.

Value

(Invisibly) A data.frame with one row per issue and some key statistics.

8.2.6 review_status

Generate a summary report for incoming Editor-in-Charge of current state of all open software-review issues.

Description

Generate a summary report for incoming Editor-in-Charge of current state of all open software-review issues.

Usage

```
review_status(open_only = TRUE, browse = TRUE, quiet = FALSE)
```

Arguments

Argument	Description
open_only	If TRUE (default), only extract information for currently open issues.
browse	If TRUE (default), open the results as a DT datatable HTML page in default browser.
quiet	If FALSE, display progress information on screen.

Value

A data.frame with one row per issue and some key statistics.

The following are links to notes on maintaining the individual components of the "ropensci-review-tools" software ecosystem.

CHAPTER

NINE

PKGSTATS

Maintenance of `pkgstats` package

CHAPTER

TEN

PKGCHECK

Maintenance of `pkgcheck` package.

10.1 Adding new checks

The procedure for adding new checks is documented in the *"Extending or modifying checks" vignette* within the `pkgcheck` package. The `roreviewapi` package which delivers the checks to rOpenSci's GitHub issues directly dumps all checks currently delivered by `pkgcheck`. See the following section for options for restricting which checks are delivered.

10.2 Controlling checks in `roreviewapi`

The addition of new checks is fairly straightforward, and described in the above section, along with corresponding links. The remainder of this section describes how the `roreviewapi` may be modified to deliver a reduced set of checks than the full set returned by `pkgcheck`.

The plumber endpoint for editor checks is entirely controlled by the `roreviewapi::editor_check()` function. The main call is via `tryCatch` to ensure any errors are captured:

```
checks <- tryCatch (pkgcheck::pkgcheck (path),
                    error = function (e) e)
```

The return object, `checks`, is a list of checks ultimately composed in the main `pkgcheck()` function definition. The easiest way to remove checks from the `roreviewpi` without modifying the underlying structure of `pkgcheck` itself is to run the checks as above, and then remove the corresponding list items. For example, the following modification would suffice to remove the `scrap` check (which checks whether a repository contains "scrap" files which should not be included) from the API endpoint:

```
checks <- tryCatch (pkgcheck::pkgcheck (path),
                    error = function (e) e)
checks$scrap <- NULL
```

When those checks are printed (via `print`) or summarised (via `summarise`), the scrap checks will not be reported on. This procedure can be used to remove checks from the `roreviewapi` results, by finding the line in the `roreviewapi::editor_check()` function where the main `pkgcheck::pkgcheck()` function is called, and then removing any checks immediately afterward by setting them to `NULL`. Those checks will then be removed from the report delivered by `roreviewpi`, and therefore by the `ropensci-review-bot`.

CHAPTER

ELEVEN

ROREVIEWAPI

This package contains the external service which the `ropensci-review-bot` calls to run package checks. The service itself is a Plumber API hosted on an external Digital Ocean server which primarily calls *the `pkgcheck()` function of the `pkgcheck` package*. The `roreviewapi` package implements post-processing routines to format the checks and deliver the results into GitHub issues. The endpoint functions are all defined in `R/plumber.R`, which is mirrored in `inst/plumber.R`.

11.1 Debugging

The endpoint includes two main debugging endpoints:

- `/log` to extract the log of submitted queries; and
- `/stdlogs` to extract `stdout` and `stderr` logs from a particular query.

11.1.1 The `/log` endpoint

The `/log` endpoint can be used to examine or debug calls directly issued by `ropensci-review-bot`. This endpoint is a `GET` method with a single parameter, `n`, specifying the number of most recent entries to return, with a default of 10. Example output from a request issued by the `ropensci-review-bot` looks like this, reformatted here for clarity:

```
INFO [2021-12-01 09:26:26]
  <sending-ip-address> "Faraday v1.8.0"
  <host-ip-address>:8000
  GET /editorcheck
    ?bot_name=ropensci-review-bot&
     issue_author=<author>&
     issue_id=<n>1&
     repo=ropensci%2Fsoftware-review&
     repourl=https%3A%2F%2Fgithub.com%2F<org>%2F<repo>&
     sender=ropensci-review-bot
  200
  1.235
```

These logs can be used to examine commands issued by the bot, which always have the machine information "Faraday". The equivalent logs for the bot can be seen by logging in to heroku, clicking on "radiant-garden", then under "More" on the upper right, clicking "Logs".

11.1.2 The /stdlogs endpoint

The `check package` command issued either manually or automatically on all new submissions returns immediately with a message, while starting a background process for the actual package checks. The `\stdlogs` endpoint is intended to help diagnose issues with this background checking process. The functions are all controlled by the `roreviewapi` package, which is in turn a plumber API providing access to the functionality of the `pkgcheck` package.

This background process dumps all `stdout` and `stderr` messages to a cached log, the contents of which can be accessed with the `\stdlogs` endpoint. This is a `GET` endpoint with the single parameter of `repourl` specifying the URL of the package being checked. Logs are cached only for the most recent calls for any one package.

These logs can be very detailed, and should provide sufficient information to diagnose most internal issues with package checking.

11.2 Manual Debugging

Some errors may arise in which a `log` entry appears to have been correctly sent, and yet no `stdlogs` are generated. These typically require manual debugging. The best approach is then to manually run the `roreviewapi` Docker container on the Digital Ocean server, and step through the code to locate the source of the problem. The general procedure for this is described in the "Debugging" vignette of the `roreviewapi` package.

The Digital Ocean droplet can be accessed via the Digital Ocean web interface, or via SSH. The latter requires a public SSH key to be registered in the droplet. Once in the droplet, the following lines will enter an R session within the Docker container:

```
docker run -it --rm roreviewapi /bin/bash
R
```

The "Debugging" vignette" then includes the following code chunk which can be stepped through to identify sources of any bugs:

```
repourl <- "https://github.com/org/repo" # replace with actual org/repo values
path <- roreviewapi::dl_gh_repo (repourl)
os <- "ubuntu"
os_release <- "20.04"
p <- roreviewapi::pkgrep_install_deps (path, os, os_release)
checks <- pkgcheck::pkgcheck(path)
out <- roreviewapi::collate_editor_check (checks)
orgrepo <- "ropensci/software-review" # or somewhere else for testing purposes
out <- roreviewapi::post_to_issue (out, orgrepo, issue_num)
```

These lines represent the main function calls within the "editor check" endpoint in the plumber function, which in turns calls the `roreviewapi::editor_check()` function. The plumber endpoints themselves should generally be bug-free, although it may be worthwhile calling the two functions prior to that call (`check_issue_template()` and `stdout_stderr_cache()`).

See the actual code of the `roreviewapi::editor_check()` function for further details, including code to handle non-default git branches. It should be possible to locate the source of most errors somewhere within one of those calls.

CHAPTER

TWELVE

ROPENSCI-REVIEW-BOT

Maintenance of `ropensci-review-bot`, built on top of the following Docker containers.

12.1 Docker Containers

12.1.1 ghcr.io/ropensci-review-tools/pkgcheck:latest

This Docker image contains most of the system libraries required to build the `roreviewapi` container. These include all libraries contained in the GitHub Ubuntu 20.04 runner, and a selection of R packages. The image is stored on the GitHub Container Registry, and rebuilt as a GitHub action as a weekly cron job (as well as on every push).

The image itself is built on top of the `rocker/r-bspm` image, for reasons explained at length in this arXiv manuscript. In short: `bspm` = Bridge to System Package Manager, and is a system to enable binary installation of system libraries as properly shared objects. Other system for installation of binary R packages in Linux, such as `rspm`, do not properly link system libraries, so that a library required by one package can not be shared by other packages. `bspm` is both very fast because of binary installation, and very robust and scalable, because system libraries are only installed once, after which they are properly shared between any additional packages which require them.

12.1.2 roreviewapi

The docker image of the `roreviewapi` repository is mainly used to pull the latest GitHub versions of the R packages contained in the `ropensci-review-tools` organisation, as well as to configure the server with the GitHub credentials of `ropensci-review-bot`.

12.2 Digital Ocean Server

All external services called by the bot are hosted on the Digital Ocean server, and defined in the `roreviewapi` package. The major endpoint of that API is the `editor_check`, called both as part of the Welcome Responder, and also able to be triggered by the `check package` command. The service is built using `docker-compose`, as controlled by the `docker-compose.yml` file, in turn built on top of the main `Dockerfile`, described above.

12.2.1 docker-compose

The docker compose actions are all contained within the `restart.sh` script within the `roreviewapi` repository. The entire service can be rebuilt and restarted by calling this script, which is also added to the `crontab` of the Digital Ocean instance, currently to update once a week.

12.2.2 Regular Maintenance

The Digital Ocean droplet should be manually updated at regular intervals (at least every few months). This requires the following two commands:

```
sudo apt-get update
sudo reboot
```

Issuing the latter will automatically close the connection to the droplet. Reconnection will only be possible after the system has successfully restarted, generally after a few minutes. Once the system has restarted, the docker-compose server will need to be restarted with the following command:

```
bash /home/shared/roreviewapi/restart.sh
```

Note that most operations within the `/home/shared` folder require `sudo`, including editing files and all `git` commands. The docker service itself *does not* require `sudo`, so the above command just calls `bash`, not `sudo bash`. Similarly, containers can be run with `docker`, and do not need `sudo docker`.

12.2.3 Managing and Reducing Disk Usage

On Digital Ocean, docker images are by default stored in `/var/lib/docker/overlay2`. This directory creates copies of every image created, and so grows continuously in size. The size of the current container can be seen with `df -h`, where the container should be `/dev/vda1`.

These images in `/var/lib/docker/overlay2` can not be removed with any `prune` command, and can only be manually cleaned. This should be done at least once every few months by manually removing all docker files and rebuilding everything from scratch. This can be done with the following commands, which remove the entire current docker installation and rebuild everything anew:

```
COMPOSE_FILE=/<path>/<to>/roreviewapi/docker-compose.yml
docker-compose -f $COMPOSE_FILE down
sudo systemctl stop docker.service
sudo su # Enter super-user mode - be very careful!!!
cd /var/lib/docker
rm -rf *
exit # Exit super-user mode
sudo systemctl start docker.service
bash /home/shared/roreviewapi/restart.sh
```

12.3 Debugging

Error messages in response to package checks can be diagnosed by following the procedures described in *the "roreviewapi" maintenance page*.

CHAPTER

THIRTEEN

GITHUB TOKENS

This section describes procedures to update and maintain GitHub tokens, which are needed for the `pkgcheck`, `pkgcheck-action`, and `roreviewapi` packages.

13.1 The "RRT_TOKEN"

"RRT" is the abbreviation of "ROpenSci Review Tools", and the "RRT_TOKEN" is the main token used in `pkgcheck` to build the Docker image needed by the external bot service, as well as the `pkgcheck-action`.

This token must be generated as a personal token by somebody with administrative rights to all repositories. That means the token must first be generated from that person's personal GitHub settings. Once (re-)generated, the token can then be copied across to the tokens in both the `pkgcheck` and `pkgcheck-action` repositories.

All token should be given durations of 60 days at most. A GitHub Actions workflow run in the `pkgcheck` repository issues a monthly notification to a specified `pkgcheck` issue for the nominated person to update the token.

13.1.1 Assigning "RRT_TOKEN" updates to a different person

The person assuming responsibility for "RRT_TOKEN" updates must:

1. Update the `pkgcheck` workflow file to ping their own GitHub name on the first line of the `body` command (currently @mpadge).
2. Respond to the notifications each time they are issued, which is currently every month. (The "0 0 1" cron specification is "hour minute day-of-month", so 00:00 on the 1st day of each month.)

13.2 Other tokens

13.2.1 roreviewapi

The `roreviewapi` repository holds one single token, "DO_IP", which stores the IP address of the Digital Ocean droplet used to host the `pkgcheck` external service called by the bot.

13.2.2 pkgcheck-action

The only token held in the pkgcheck-action repository is the "RRT_TOKEN" described above.

13.2.3 pkgcheck

The pkgcheck repository holds the following tokens:

- DOCKER_USERNAME & DOCKER_PASSWORD: Currently set for private account of @mpadge, but could easily be modified to other values. Modification would then require updating the first line of the Dockerfile in the roreviewapi repository to update the source of the base Docker image used to build that container.
- NETLIFY_SITE_ID & NETLIFY_TOKEN: Currently not used.
- UNAME: Currently not used.

No other tokens in pkgcheck are currently used.

Finally, the following section describes components of our system used to construct and maintain badges on GitHub README pages and package documentation.

CHAPTER

FOURTEEN

BADGES

This section describes procedures to create, maintain, and update badges which ultimately appear on each repositories README page.

14.1 Source for badges

The source "svg" files for all badges are held in the "svgs" directory of github.com/ropensci-org/badges. New badges needs to be created there, generally by copying an existing badge and then editing the "svg" file. These files should be edited in a code (text) editor, and *not* in an image editor.

14.1.1 Statistical Software Review Badges

Badges for statistics packages include the version number of the *Statistical Software Standards* current at the time of package acceptance, coloured according to the "grade" of standards compliance (either bronze, silver, or gold). They look like this:

Each new version of the standards requires three corresponding badges to be added to the "svgs" directory of the "badges" repository, one for each colour. This can be done by simply copying one of the previous versions, and replacing the version numbers on the last two lines of each file with updated ones.

14.2 The 'badges' server and repository

The repository holding the source badges (github.com/ropensci-org/badges) serves the primary function of assigning an appropriate badge to each rOpenSci repository. These badges are served by our "badge server" which is deployed in the last step of the GitHub Action in that repository. The badge for each repository is uniquely identified by the issue number of the software-review repository, for example 'badges.ropensci.org/222_stats.svg' for the {epubr} package reviewed in issue number 222, which looks like this:

The rOpenSci package reviewed in issue number XYZ can then display the appropriate badge by simply linking to 'https://badges.ropensci.org/XYZ_status.svg'.

14.2.1 The badge script

The server providing the badges reads them directly from the "pkgsvgs" directory of the "gh-pages" branch of that repository. This directory is populated by the main script, `update_badges.rb`. This script is called from the GitHub Action, and copies the appropriate badges from the source directory ("svgs") in the main branch to the "pkgsvgs" directory in the gh-pages branch, renaming each according to the review issue number of each package.

14.3 Constructing and implementing new badges

The following steps describe the procedure which must be followed to construct and implement new badges.

1. Design a new badge and save it in the "svgs" folder of the badges repository.
2. Design a corresponding GitHub "approved" issue label in the main software-review repository. This label must begin with "6/approved-", followed by the desired new text. Note the slash after "6" is forward, not backward!
3. The bot then needs to be modified to recognise any new badges, like this code which processes statistics badges. That will then ensure that the `approve` command will also add the corresponding label to the issue thread.
4. Modify the `update_badges.rb` script in the badges repository to recognise the new labels. The value of these new values will then generally be in the first field of the `iss_peer_rev_files` object, currently called "color". The remainder of that script will then also need modifying to process entires with `color` values but no `version` values…

After updating the script, the new badges should be automatically deployed and assigned to the appropriate review issues.