

Package ‘usethis’

May 8, 2026

Title Automate Package and Project Setup

Version 3.2.1

Description Automate package and project setup tasks that are otherwise performed manually. This includes setting up unit testing, test coverage, continuous integration, Git, 'GitHub', licenses, 'Rcpp', 'RStudio' projects, and more.

License MIT + file LICENSE

URL <https://usethis.r-lib.org>, <https://github.com/r-lib/usethis>

BugReports <https://github.com/r-lib/usethis/issues>

Depends R (>= 4.1)

Imports cli (>= 3.0.1), clipr (>= 0.3.0), crayon, curl (>= 2.7), desc (>= 1.4.2), fs (>= 1.3.0), gert (>= 1.4.1), gh (>= 1.2.1), glue (>= 1.3.0), jsonlite, lifecycle (>= 1.0.0), purrr, rappdirs, rlang (>= 1.1.0), rprojroot (>= 2.1.1), rstudioapi, stats, tools, utils, whisker, withr (>= 2.3.0), yaml

Suggests covr, knitr, magick, pkgload (>= 1.3.2.1), quarto (>= 1.5.1), rmarkdown, roxygen2 (>= 7.1.2), spelling (>= 1.2), testthat (>= 3.1.8)

Config/Needs/website r-lib/asciicast, tidyverse/tidytemplate, xml2

Config/testthat/edition 3

Config/testthat/parallel TRUE

Config/testthat/start-first github-actions, release

Config/usethis/last-upkeep 2025-04-22

Encoding UTF-8

Language en-US

RoxygenNote 7.3.3

NeedsCompilation no

Author Hadley Wickham [aut] (ORCID: <<https://orcid.org/0000-0003-4757-117X>>),
Jennifer Bryan [aut, cre] (ORCID:
<<https://orcid.org/0000-0002-6983-2759>>),

Malcolm Barrett [aut] (ORCID: <<https://orcid.org/0000-0003-0299-5825>>),
 Andy Teucher [aut] (ORCID: <<https://orcid.org/0000-0002-7840-692X>>),
 Posit Software, PBC [cph, fnd] (ROR: <<https://ror.org/03wc8by49>>)

Maintainer Jennifer Bryan <jenny@posit.co>

Repository CRAN

Date/Publication 2025-09-06 05:10:02 UTC

Contents

| | |
|-------------------------------|----|
| badges | 4 |
| browse-this | 5 |
| create_from_github | 7 |
| create_package | 9 |
| edit | 11 |
| git-default-branch | 12 |
| github-token | 14 |
| git_protocol | 16 |
| git_sitrep | 17 |
| git_vaccinate | 18 |
| issue-this | 18 |
| licenses | 19 |
| proj_activate | 20 |
| proj_sitrep | 21 |
| proj_utils | 22 |
| pull-requests | 24 |
| rename_files | 27 |
| rprofile-helper | 28 |
| ui_silence | 28 |
| usethis_options | 29 |
| use_addin | 30 |
| use_air | 31 |
| use_author | 32 |
| use_blank_slate | 33 |
| use_build_ignore | 34 |
| use_citation | 34 |
| use_code_of_conduct | 35 |
| use_coverage | 35 |
| use_cpp11 | 36 |
| use_cran_comments | 36 |
| use_data | 37 |
| use_data_table | 38 |
| use_description | 38 |
| use_directory | 40 |
| use_git | 40 |
| use_github | 41 |
| use_github_action | 43 |
| use_github_file | 45 |

| | |
|-----------------------------------|----|
| use_github_labels | 46 |
| use_github_links | 48 |
| use_github_pages | 49 |
| use_github_release | 50 |
| use_gitlab_ci | 50 |
| use_git_config | 51 |
| use_git_hook | 52 |
| use_git_ignore | 53 |
| use_git_remote | 53 |
| use_import_from | 54 |
| use_jenkins | 55 |
| use_lifecycle | 56 |
| use_logo | 56 |
| use_make | 57 |
| use_namespace | 57 |
| use_news_md | 58 |
| use_package | 58 |
| use_package_doc | 59 |
| use_pipe | 60 |
| use_pkgdown | 60 |
| use_r | 61 |
| use_rcpp | 63 |
| use_readme_rmd | 63 |
| use_release_issue | 64 |
| use_revdep | 65 |
| use_rmarkdown_template | 66 |
| use_roxygen_md | 67 |
| use_rstudio | 67 |
| use_rstudio_preferences | 68 |
| use_spell_check | 68 |
| use_standalone | 69 |
| use_template | 70 |
| use_testthat | 71 |
| use_test_helper | 72 |
| use_tibble | 73 |
| use_tidy_github_actions | 74 |
| use_tidy_thanks | 76 |
| use_tutorial | 77 |
| use_upkeep_issue | 78 |
| use_version | 78 |
| use_vignette | 79 |
| zip-utils | 80 |

Description

These helpers produce the markdown text you need in your README to include badges that report information, such as the CRAN version or test coverage, and link out to relevant external resources. To allow badges to be added automatically, ensure your badge block starts with a line containing only `<!-- badges: start -->` and ends with a line containing only `<!-- badges: end -->`. The templates used by `use_readme_md()` and `use_readme_rmd()` include this block.

Usage

```
use_badge(badge_name, href, src)

use_cran_badge()

use_bioc_badge()

use_lifecycle_badge(stage)

use_binder_badge(ref = git_default_branch(), urlpath = NULL)

use_r_universe_badge(repo_spec = NULL)

use_posit_cloud_badge(url)
```

Arguments

| | |
|-------------------------|---|
| <code>badge_name</code> | Badge name. Used in error message and alt text |
| <code>href, src</code> | Badge link and image src |
| <code>stage</code> | Stage of the package lifecycle. One of "experimental", "stable", "superseded", or "deprecated". |
| <code>ref</code> | A Git branch, tag, or SHA |
| <code>urlpath</code> | An optional <code>urlpath</code> component to add to the link, e.g. "rstudio" to open an RStudio IDE instead of a Jupyter notebook. See the binder documentation for additional examples. |
| <code>repo_spec</code> | Optional GitHub repo specification in this form: owner/repo. This can usually be inferred from the GitHub remotes of active project. |
| <code>url</code> | A link to an existing Posit Cloud project. See the Posit Cloud documentation for details on how to set project access and obtain a project link. |

Details

- `use_badge()`: a general helper used in all badge functions
- `use_bioc_badge()`: badge indicates **BioConductor build status**
- `use_cran_badge()`: badge indicates what version of your package is available on CRAN, powered by <https://www.r-pkg.org>
- `use_lifecycle_badge()`: badge declares the developmental stage of a package according to <https://lifecycle.r-lib.org/articles/stages.html>.
- `use_r_universe_badge()`: **[Experimental]** badge indicates what version of your package is available on **R-universe** . It is assumed that you have already completed the **necessary R-universe setup**.
- `use_binder_badge()`: badge indicates that your repository can be launched in an executable environment on <https://mybinder.org/>
- `use_posit_cloud_badge()`: badge indicates that your repository can be launched in a **Posit Cloud** project
- `use_rscloud_badge()`: **[Deprecated]** Use `use_posit_cloud_badge()` instead.

See Also

`use_github_action()` helps with the setup of various continuous integration workflows, some of which will call these specialized badge helpers.

Examples

```
## Not run:
use_cran_badge()
use_lifecycle_badge("stable")

## End(Not run)
```

browse-this

Visit important project-related web pages

Description

These functions take you to various web pages associated with a project (often, an R package) and return the target URL(s) invisibly. To form these URLs we consult:

- Git remotes configured for the active project that appear to be hosted on a GitHub deployment
- DESCRIPTION file for the active project or the specified package. The DESCRIPTION file is sought first in the local package library and then on CRAN.
- Fixed templates:
 - Circle CI: <https://circleci.com/gh/{OWNER}/{PACKAGE}>
 - CRAN landing page: <https://cran.r-project.org/package={PACKAGE}>
 - GitHub mirror of a CRAN package: <https://github.com/cran/{PACKAGE}> Templated URLs aren't checked for existence, so there is no guarantee there will be content at the destination.

Usage

```
browse_package(package = NULL)

browse_project()

browse_github(package = NULL)

browse_github_issues(package = NULL, number = NULL)

browse_github_pulls(package = NULL, number = NULL)

browse_github_actions(package = NULL)

browse_circleci(package = NULL)

browse_cran(package = NULL)
```

Arguments

| | |
|---------|---|
| package | Name of package. If NULL, the active project is targeted, regardless of whether it's an R package or not. |
| number | Optional, to specify an individual GitHub issue or pull request. Can be a number or "new". |

Details

- `browse_package()`: Assembles a list of URLs and lets user choose one to visit in a web browser. In a non-interactive session, returns all discovered URLs.
- `browse_project()`: Thin wrapper around `browse_package()` that always targets the active usethis project.
- `browse_github()`: Visits a GitHub repository associated with the project. In the case of a fork, you might be asked to specify if you're interested in the source repo or your fork.
- `browse_github_issues()`: Visits the GitHub Issues index or one specific issue.
- `browse_github_pulls()`: Visits the GitHub Pull Request index or one specific pull request.
- `browse_circleci()`: Visits the project's page on [Circle CI](#).
- `browse_cran()`: Visits the package on CRAN, via the canonical URL.

Examples

```
# works on the active project
# browse_project()

browse_package("httr")
browse_github("gh")
browse_github_issues("fs")
browse_github_issues("fs", 1)
browse_github_pulls("curl")
```

```
browse_github_pulls("curl", 183)
browse_cran("MASS")
```

create_from_github *Create a project from a GitHub repo*

Description

Creates a new local project and Git repository from a repo on GitHub, by either cloning or **fork-and-cloning**. In the fork-and-clone case, `create_from_github()` also does additional remote and branch setup, leaving you in the perfect position to make a pull request with `pr_init()`, one of several [functions for working with pull requests](#).

`create_from_github()` works best when your GitHub credentials are discoverable. See below for more about authentication.

Usage

```
create_from_github(
  repo_spec,
  destdir = NULL,
  fork = NA,
  rstudio = NULL,
  open = rlang::is_interactive(),
  protocol = git_protocol(),
  host = NULL
)
```

Arguments

| | |
|-----------|---|
| repo_spec | A string identifying the GitHub repo in one of these forms: <ul style="list-style-type: none"> • Plain OWNER/REPO spec • Browser URL, such as "https://github.com/OWNER/REPO" • HTTPS Git URL, such as "https://github.com/OWNER/REPO.git" • SSH Git URL, such as "git@github.com:OWNER/REPO.git" |
| destdir | Destination for the new folder, which will be named according to the REPO extracted from repo_spec. Defaults to the location stored in the global option <code>usethis.destdir</code> , if defined, or to the user's Desktop or similarly conspicuous place otherwise. |
| fork | If FALSE, we clone repo_spec. If TRUE, we fork repo_spec, clone that fork, and do additional setup favorable for future pull requests: <ul style="list-style-type: none"> • The source repo, repo_spec, is configured as the upstream remote, using the indicated protocol. • The local DEFAULT branch is set to track upstream/DEFAULT, where DEFAULT is typically main or master. It is also immediately pulled, to cover the case of a pre-existing, out-of-date fork. |

| | |
|----------|---|
| | If fork = NA (the default), we check your permissions on repo_spec. If you can push, we set fork = FALSE, If you cannot, we set fork = TRUE. |
| rstudio | Initiate an RStudio Project ? Defaults to TRUE if in an RStudio session and project has no pre-existing .Rproj file. Defaults to FALSE otherwise (but note that the cloned repo may already be an RStudio Project, i.e. may already have a .Rproj file). |
| open | If TRUE, activates the new project: <ul style="list-style-type: none"> • If using RStudio or Positron, the new project is opened in a new session, window, or browser tab, depending on the product (RStudio or Positron) and context (desktop or server). • Otherwise, the working directory and active project of the current R session are changed to the new project. |
| protocol | One of "https" or "ssh" |
| host | GitHub host to target, passed to the .api_url argument of <code>gh::gh()</code> . If repo_spec is a URL, host is extracted from that. If unspecified, gh defaults to "https://api.github.com", although gh's default can be customised by setting the GITHUB_API_URL environment variable. For a hypothetical GitHub Enterprise instance, either "https://github.acme.com/api/v3" or "https://github.acme.com" is acceptable. |

Git/GitHub Authentication

Many usethis functions, including those documented here, potentially interact with GitHub in two different ways:

- Via the GitHub REST API. Examples: create a repo, a fork, or a pull request.
- As a conventional Git remote. Examples: clone, fetch, or push.

Therefore two types of auth can happen and your credentials must be discoverable. Which credentials do we mean?

- A GitHub personal access token (PAT) must be discoverable by the gh package, which is used for GitHub operations via the REST API. See `gh_token_help()` for more about getting and configuring a PAT.
- If you use the HTTPS protocol for Git remotes, your PAT is also used for Git operations, such as `git push`. Useshis uses the gert package for this, so the PAT must be discoverable by gert. Generally gert and gh will discover and use the same PAT. This ability to "kill two birds with one stone" is why HTTPS + PAT is our recommended auth strategy for those new to Git and GitHub and PRs.
- If you use SSH remotes, your SSH keys must also be discoverable, in addition to your PAT. The public key must be added to your GitHub account.

Git/GitHub credential management is covered in a dedicated article: [Managing Git\(Hub\) Credentials](#)

See Also

- [use_github\(\)](#) to go the opposite direction, i.e. create a GitHub repo from your local repo
- [git_protocol\(\)](#) for background on protocol (HTTPS vs SSH)
- [use_course\(\)](#) to download a snapshot of all files in a GitHub repo, without the need for any local or remote Git operations

Examples

```
## Not run:
create_from_github("r-lib/usethis")

# repo_spec can be a URL
create_from_github("https://github.com/r-lib/usethis")

# a URL repo_spec also specifies the host (e.g. GitHub Enterprise instance)
create_from_github("https://github.acme.com/OWNER/REPO")

## End(Not run)
```

| | |
|----------------|------------------------------------|
| create_package | <i>Create a package or project</i> |
|----------------|------------------------------------|

Description

These functions create an R project:

- `create_package()` creates an R package.
- `create_project()` creates a non-package project, i.e. a data analysis project.
- **[Experimental]** `create_quarto_project()` creates a Quarto project. It is a simplified convenience wrapper around `quarto::quarto_create_project()`, which you should call directly for more advanced usage.

These functions work best when creating a project *de novo*, but `create_package()` and `create_project()` can be called on an existing project; you will be asked before any existing files are changed.

Usage

```
create_package(
  path,
  fields = list(),
  rstudio = rstudioapi::isAvailable(),
  roxygen = TRUE,
  check_name = TRUE,
  open = rlang::is_interactive()
)

create_project(
```

```

  path,
  rstudio = rstudioapi::isAvailable(),
  open = rlang::is_interactive()
)

create_quarto_project(
  path,
  type = "default",
  rstudio = rstudioapi::isAvailable(),
  open = rlang::is_interactive()
)

```

Arguments

| | |
|------------|---|
| path | A path. If it exists, it is used. If it does not exist, it is created, provided that the parent path exists. |
| fields | A named list of fields to add to DESCRIPTION, potentially overriding default values. See use_description() for how you can set personalized defaults using package options. |
| rstudio | If TRUE, calls use_rstudio() to make the new package or project into an RStudio Project . If FALSE, the goal is to ensure that the directory can be recognized as a project by, for example, the here package. If the project is neither an R package nor a Quarto project, a sentinel <code>.here</code> file is placed to mark the project root. |
| roxygen | Do you plan to use roxygen2 to document your package? |
| check_name | Whether to check if the name is valid for CRAN and throw an error if not. |
| open | If TRUE, activates the new project: <ul style="list-style-type: none"> • If using RStudio or Positron, the new project is opened in a new session, window, or browser tab, depending on the product (RStudio or Positron) and context (desktop or server). • Otherwise, the working directory and active project of the current R session are changed to the new project. |
| type | The type of Quarto project to create. See <code>?quarto::quarto_create_project</code> for the most up-to-date list, but "website", "blog", "book", and "manuscript" are common choices. |

Value

Path to the newly created project or package, invisibly.

See Also

[create_tidy_package\(\)](#) is a convenience function that extends `create_package()` by immediately applying as many of the tidyverse development conventions as possible.

`edit`*Open configuration files*

Description

- `edit_r_profile()` opens `.Rprofile`
- `edit_r_environ()` opens `.Renviron`
- `edit_r_makevars()` opens `.R/Makevars`
- `edit_git_config()` opens `.gitconfig` or `.git/config`
- `edit_git_ignore()` opens global (user-level) `gitignore` file and ensures its path is declared in your global Git config.
- `edit_pkgdown_config` opens the `pkgdown` YAML configuration file for the current Project.
- `edit_rstudio_snippets()` opens RStudio's snippet config for the given type.
- `edit_rstudio_prefs()` opens [RStudio's preference file](#).

Usage

```
edit_r_profile(scope = c("user", "project"))

edit_r_environ(scope = c("user", "project"))

edit_r_buildignore()

edit_r_makevars(scope = c("user", "project"))

edit_rstudio_snippets(
  type = c("r", "markdown", "c_cpp", "css", "html", "java", "javascript", "python",
          "sql", "stan", "tex", "yaml")
)

edit_rstudio_prefs()

edit_git_config(scope = c("user", "project"))

edit_git_ignore(scope = c("user", "project"))

edit_pkgdown_config()
```

Arguments

| | |
|--------------------|---|
| <code>scope</code> | Edit globally for the current user , or locally for the current project |
| <code>type</code> | Snippet type (case insensitive text). |

Details

The `edit_r_*`() functions consult R's notion of user's home directory. The `edit_git_*`() functions (and **usethis** in general) inherit home directory behaviour from the **fs** package, which differs from R itself on Windows. The **fs** default is more conventional in terms of the location of user-level Git config files. See `fs::path_home()` for more details.

Files created by `edit_rstudio_snippets()` will *mask*, not supplement, the built-in default snippets. If you like the built-in snippets, copy them and include with your custom snippets.

Value

Path to the file, invisibly.

| | |
|---------------------------------|--|
| <code>git-default-branch</code> | <i>Get or set the default Git branch</i> |
|---------------------------------|--|

Description

The `git_default_branch*`() functions put some structure around the somewhat fuzzy (but definitely real) concept of the default branch. In particular, they support new conventions around the Git default branch name, globally or in a specific project / Git repository.

Usage

```
git_default_branch()

git_default_branch_configure(name = "main")

git_default_branch_rediscover(current_local_default = NULL)

git_default_branch_rename(from = NULL, to = "main")
```

Arguments

| | |
|------------------------------------|---|
| <code>name</code> | Default name for the initial branch in new Git repositories. |
| <code>current_local_default</code> | Name of the local branch that is currently functioning as the default branch. If unspecified, this can often be inferred. |
| <code>from</code> | Name of the branch that is currently functioning as the default branch. |
| <code>to</code> | New name for the default branch. |

Value

Name of the default branch.

Background on the default branch

Technically, Git has no official concept of the default branch. But in reality, almost all Git repos have an *effective default branch*. If there's only one branch, this is it! It is the branch that most bug fixes and features get merged in to. It is the branch you see when you first visit a repo on a site such as GitHub. On a Git remote, it is the branch that HEAD points to.

Historically, `master` has been the most common name for the default branch, but `main` is an increasingly popular choice.

`git_default_branch_configure()`

This configures `init.defaultBranch` at the global (a.k.a user) level. This setting determines the name of the branch that gets created when you make the first commit in a new Git repo. `init.defaultBranch` only affects the local Git repos you create in the future.

`git_default_branch()`

This figures out the default branch of the current Git repo, integrating information from the local repo and, if applicable, the upstream or `origin` remote. If there is a local vs. remote mismatch, `git_default_branch()` throws an error with advice to call `git_default_branch_rediscover()` to repair the situation.

For a remote repo, the default branch is the branch that HEAD points to.

For the local repo, if there is only one branch, that must be the default! Otherwise we try to identify the relevant local branch by looking for specific branch names, in this order:

- whatever the default branch of upstream or `origin` is, if applicable
- `main`
- `master`
- the value of the Git option `init.defaultBranch`, with the usual deal where a local value, if present, takes precedence over a global (a.k.a. user-level) value

`git_default_branch_rediscover()`

This consults an external authority – specifically, the remote **source repo** on GitHub – to learn the default branch of the current project / repo. If that doesn't match the apparent local default branch (for example, the project switched from `master` to `main`), we do the corresponding branch renaming in your local repo and, if relevant, in your fork.

See <https://happygitwithr.com/common-remote-setups.html> for more about GitHub remote configurations and, e.g., what we mean by the source repo. This function works for the configurations "ours", "fork", and "theirs".

`git_default_branch_rename()`

Note: this only works for a repo that you effectively own. In terms of GitHub, you must own the **source repo** personally or, if organization-owned, you must have admin permission on the **source repo**.

This renames the default branch in the **source repo** on GitHub and then calls `git_default_branch_rediscover()`, to make any necessary changes in the local repo and, if relevant, in your personal fork.

See <https://happygitwithr.com/common-remote-setups.html> for more about GitHub remote configurations and, e.g., what we mean by the source repo. This function works for the configurations "ours", "fork", and "no_github".

Regarding "no_github": Of course, this function does what you expect for a local repo with no GitHub remotes, but that is not the primary use case.

Examples

```
## Not run:
git_default_branch()

## End(Not run)
## Not run:
git_default_branch_configure()

## End(Not run)
## Not run:
git_default_branch_rediscover()

# you can always explicitly specify the local branch that's been playing the
# role of the default
git_default_branch_rediscover("unconventional_default_branch_name")

## End(Not run)
## Not run:
git_default_branch_rename()

# you can always explicitly specify one or both branch names
git_default_branch_rename(from = "this", to = "that")

## End(Not run)
```

github-token

Get help with GitHub personal access tokens

Description

A **personal access token** (PAT) is needed for certain tasks usethis does via the GitHub API, such as creating a repository, a fork, or a pull request. If you use HTTPS remotes, your PAT is also used when interacting with GitHub as a conventional Git remote. These functions help you get and manage your PAT:

- `gh_token_help()` guides you through token troubleshooting and setup.
- `create_github_token()` opens a browser window to the GitHub form to generate a PAT, with suggested scopes pre-selected. It also offers advice on storing your PAT.
- `gitcreds::gitcreds_set()` helps you register your PAT with the Git credential manager used by your operating system. Later, other packages, such as usethis, gert, and gh can automatically retrieve that PAT and use it to work with GitHub on your behalf.

Usually, the first time the PAT is retrieved in an R session, it is cached in an environment variable, for easier reuse for the duration of that R session. After initial acquisition and storage, all of this should happen automatically in the background. GitHub is encouraging the use of PATs that expire after, e.g., 30 days, so prepare yourself to re-generate and re-store your PAT periodically.

Git/GitHub credential management is covered in a dedicated article: [Managing Git\(Hub\) Credentials](#)

Usage

```
create_github_token(  
  scopes = c("repo", "user", "gist", "workflow"),  
  description = "DESCRIBE THE TOKEN'S USE CASE",  
  host = NULL  
)  
  
gh_token_help(host = NULL)
```

Arguments

| | |
|-------------|---|
| scopes | Character vector of token scopes, pre-selected in the web form. Final choices are made in the GitHub form. Read more about GitHub API scopes at https://docs.github.com/apps/building-oauth-apps/understanding-scopes-for-oauth-apps/ . |
| description | Short description or nickname for the token. You might (eventually) have multiple tokens on your GitHub account and a label can help you keep track of what each token is for. |
| host | GitHub host to target, passed to the <code>.api_url</code> argument of <code>gh::gh()</code> . If unspecified, <code>gh</code> defaults to "https://api.github.com", although <code>gh</code> 's default can be customised by setting the <code>GITHUB_API_URL</code> environment variable. For a hypothetical GitHub Enterprise instance, either "https://github.acme.com/api/v3" or "https://github.acme.com" is acceptable. |

Details

`create_github_token()` has previously gone by some other names: `browse_github_token()` and `browse_github_pat()`.

Value

Nothing

See Also

[gh::gh_whoami\(\)](#) for information on an existing token and `gitcreds::gitcreds_set()` and `gitcreds::gitcreds_get()` for a secure way to store and retrieve your PAT.

Examples

```
## Not run:
create_github_token()

## End(Not run)
## Not run:
gh_token_help()

## End(Not run)
```

git_protocol

See or set the default Git protocol

Description

Git operations that address a remote use a so-called "transport protocol". usethis supports HTTPS and SSH. The protocol dictates the Git URL format used when usethis needs to configure the first GitHub remote for a repo:

- protocol = "https" implies `https://github.com/<OWNER>/<REPO>.git`
- protocol = "ssh" implies `git@github.com:<OWNER>/<REPO>.git`

Two helper functions are available:

- `git_protocol()` reveals the protocol "in force". As of usethis v2.0.0, this defaults to "https". You can change this for the duration of the R session with `use_git_protocol()`. Change the default for all R sessions with code like this in your `.Rprofile` (easily editable via [edit_r_profile\(\)](#)):

```
options(usethis.protocol = "ssh")
```

- `use_git_protocol()` sets the Git protocol for the current R session

This protocol only affects the Git URL for newly configured remotes. All existing Git remote URLs are always respected, whether HTTPS or SSH.

Usage

```
git_protocol()

use_git_protocol(protocol)
```

Arguments

protocol One of "https" or "ssh"

Value

The protocol, either "https" or "ssh"

Examples

```
## Not run:
git_protocol()

use_git_protocol("ssh")
git_protocol()

use_git_protocol("https")
git_protocol()

## End(Not run)
```

git_sitrep

Git/GitHub sitrep

Description

Get a situation report on your current Git/GitHub status. Useful for diagnosing problems. The default is to report all values; provide values for `tool` or `scope` to be more specific.

Usage

```
git_sitrep(tool = c("git", "github"), scope = c("user", "project"))
```

Arguments

| | |
|--------------------|---|
| <code>tool</code> | Report for git , or github |
| <code>scope</code> | Report globally for the current user , or locally for the current project |

Examples

```
## Not run:
# report all
git_sitrep()

# report git for current user
git_sitrep("git", "user")

## End(Not run)
```

| | |
|---------------|---|
| git_vaccinate | <i>Vaccinate your global gitignore file</i> |
|---------------|---|

Description

Adds `.Rproj.user`, `.Rhistory`, `.Rdata`, `.httr-oauth`, `.DS_Store`, and `.quarto` to your global (a.k.a. user-level) `.gitignore`. This is good practice as it decreases the chance that you will accidentally leak credentials to GitHub. `git_vaccinate()` also tries to detect and fix the situation where you have a global gitignore file, but it's missing from your global Git config.

Usage

```
git_vaccinate()
```

| | |
|------------|----------------------------------|
| issue-this | <i>Helpers for GitHub issues</i> |
|------------|----------------------------------|

Description

The `issue_*` family of functions allows you to perform common operations on GitHub issues from within R. They're designed to help you efficiently deal with large numbers of issues, particularly motivated by the challenges faced by the tidyverse team.

- `issue_close_community()` closes an issue, because it's not a bug report or feature request, and points the author towards Posit Community as a better place to discuss usage (<https://forum.posit.co>).
- `issue_reprex_needed()` labels the issue with the "reprex" label and gives the author some advice about what is needed.

Usage

```
issue_close_community(number, reprex = FALSE)
```

```
issue_reprex_needed(number)
```

Arguments

| | |
|--------|------------------------------------|
| number | Issue number |
| reprex | Does the issue also need a reprex? |

Saved replies

Unlike GitHub's "saved replies", these functions can:

- Be shared between people
- Perform other actions, like labelling, or closing
- Have additional arguments
- Include randomness (like friendly gifs)

Examples

```
## Not run:
issue_close_community(12, repress = TRUE)

issue_repress_needed(241)

## End(Not run)
```

licenses

License a package

Description

Adds the necessary infrastructure to declare your package as licensed with one of these popular open source licenses:

Permissive:

- **MIT**: simple and permissive.
- **Apache 2.0**: MIT + provides patent protection.

Copyleft:

- **GPL v2**: requires sharing of improvements.
- **GPL v3**: requires sharing of improvements.
- **AGPL v3**: requires sharing of improvements.
- **LGPL v2.1**: requires sharing of improvements.
- **LGPL v3**: requires sharing of improvements.

Creative commons licenses appropriate for data packages:

- **CC0**: dedicated to public domain.
- **CC-BY**: Free to share and adapt, must give appropriate credit.

See <https://choosealicense.com> for more details and other options.

Alternatively, for code that you don't want to share with others, `use_proprietary_license()` makes it clear that all rights are reserved, and the code is not open source.

Usage

```
use_mit_license(copyright_holder = NULL)

use_gpl_license(version = 3, include_future = TRUE)

use_agpl_license(version = 3, include_future = TRUE)

use_lgpl_license(version = 3, include_future = TRUE)
```

```
use_apache_license(version = 2, include_future = TRUE)
```

```
use_cc0_license()
```

```
use_ccby_license()
```

```
use_proprietary_license(copyright_holder)
```

Arguments

copyright_holder

Name of the copyright holder or holders. This defaults to "{package name} authors"; you should only change this if you use a CLA to assign copyright to a single entity.

version

License version. This defaults to latest version all licenses.

include_future

If TRUE, will license your package under the current and any potential future versions of the license. This is generally considered to be good practice because it means your package will automatically include "bug" fixes in licenses.

Details

CRAN does not permit you to include copies of standard licenses in your package, so these functions save the license as LICENSE.md and add it to .Rbuildignore.

See Also

For more details, refer to the the [license chapter](#) in *R Packages*.

proj_activate

Activate a project

Description

Activates a project in the usethis, R session, and (if relevant) RStudio senses. If you are in RStudio, this will open a new RStudio session. If not, it will change the working directory and [active project](#).

Usage

```
proj_activate(path)
```

Arguments

path

Project directory

Details

- If using RStudio desktop, the project is opened in a new session.
 - If using Positron, the project is opened in a new window.
 - If using RStudio or Positron on a server, the project is opened in a new browser tab.
 - Otherwise, the working directory and active project is changed in the current R session.

Value

Single logical value indicating if current session is modified.

proj_sitrep

Report working directory and usethis/RStudio project

Description

proj_sitrep() reports

- current working directory
- the active usethis project
- the active RStudio Project

Call this function if things seem weird and you're not sure what's wrong or how to fix it. Usually, all three of these should coincide (or be unset) and proj_sitrep() provides suggested commands for getting back to this happy state.

Usage

```
proj_sitrep()
```

Value

A named list, with S3 class `sitrep` (for printing purposes), reporting current working directory, active usethis project, and active RStudio Project

See Also

Other project functions: [proj_utils](#)

Examples

```
proj_sitrep()
```

Description

Most `use_*()` functions act on the **active project**. If it is unset, `usethis` uses `rprojroot` to find the project root of the current working directory. It establishes the project root by looking for signs such as:

- a `.here` file
- an RStudio Project, i.e. a `.Rproj` file
- an R package, i.e. a `DESCRIPTION` file
- a Git repository
- a Positron or VS Code workspace, i.e. a `.vscode/settings.json` file
- a Quarto project, i.e. a `_quarto.yml` file
- an `renv` project, i.e. a `renv.lock` file

`usethis` then stores the active project for use for the remainder of the session.

In general, end user scripts should not contain direct calls to `usethis::proj_*` utility functions. They are internal functions that are exported for occasional interactive use or use in packages that extend `usethis`. End user code should call `here::here()` or other functions from the `here` or `rprojroot` packages to programmatically detect a project and build paths within it.

If you are puzzled why a path (usually the current working directory) does *not* appear to be inside project, it can be helpful to call `here::dr_here()` to get much more verbose feedback.

Usage

```
proj_get()
```

```
proj_set(path = ".", force = FALSE)
```

```
proj_path(..., ext = "")
```

```
with_project(  
  path = ".",  
  code,  
  force = FALSE,  
  setwd = TRUE,  
  quiet = getOption("usethis.quiet", default = FALSE)  
)
```

```
local_project(  
  path = ".",  
  force = FALSE,  
  setwd = TRUE,
```

```

    quiet = getOption("usethis.quiet", default = FALSE),
    .local_envir = parent.frame()
  )

```

Arguments

| | |
|--------------|--|
| path | Path to set. This path should exist or be NULL. |
| force | If TRUE, use this path without checking the usual criteria for a project. Use sparingly! The main application is to solve a temporary chicken-egg problem: you need to set the active project in order to add project-signalling infrastructure, such as initialising a Git repo or adding a DESCRIPTION file. |
| ... | character vectors, if any values are NA, the result will also be NA. The paths follow the recycling rules used in the tibble package, namely that only length 1 arguments are recycled. |
| ext | An optional extension to append to the generated path. |
| code | Code to run with temporary active project |
| setwd | Whether to also temporarily set the working directory to the active project, if it is not NULL |
| quiet | Whether to suppress user-facing messages, while operating in the temporary active project |
| .local_envir | The environment to use for scoping. Defaults to current execution environment. |

Functions

- `proj_get()`: Retrieves the active project and, if necessary, attempts to set it in the first place.
- `proj_set()`: Sets the active project.
- `proj_path()`: Builds paths within the active project returned by `proj_get()`. Thin wrapper around `fs::path()`.
- `with_project()`: Runs code with a temporary active project and, optionally, working directory. It is an example of the `with_*`() functions in `withr`.
- `local_project()`: Sets an active project and, optionally, working directory until the current execution environment goes out of scope, e.g. the end of the current function or test. It is an example of the `local_*`() functions in `withr`.

See Also

Other project functions: [proj_sitrep\(\)](#)

Examples

```

## Not run:
## see the active project
proj_get()

## manually set the active project
proj_set("path/to/target/project")

```

```
## build a path within the active project (both produce same result)
proj_path("R/foo.R")
proj_path("R", "foo", ext = "R")

## build a path within SOME OTHER project
with_project("path/to/some/other/project", proj_path("blah.R"))

## convince yourself that with_project() temporarily changes the project
with_project("path/to/some/other/project", print(proj_sitrep()))

## End(Not run)
```

pull-requests

Helpers for GitHub pull requests

Description

The `pr_*` family of functions is designed to make working with GitHub pull requests (PRs) as painless as possible for both contributors and package maintainers.

To use the `pr_*` functions, your project must be a Git repo and have one of these GitHub remote configurations:

- "ours": You can push to the GitHub remote configured as `origin` and it's not a fork.
- "fork": You can push to the GitHub remote configured as `origin`, it's a fork, and its parent is configured as `upstream`. `origin` points to your **personal** copy and `upstream` points to the **source repo**.

"Ours" and "fork" are two of several GitHub remote configurations examined in [Common remote setups](#) in *Happy Git and GitHub for the useR*.

The [Pull Request Helpers](#) article walks through the process of making a pull request with the `pr_*` functions.

The `pr_*` functions also use your Git/GitHub credentials to carry out various remote operations; see below for more about auth. The `pr_*` functions also proactively check for agreement re: the default branch in your local repo and the source repo. See [git_default_branch\(\)](#) for more.

Usage

```
pr_init(branch)

pr_resume(branch = NULL)

pr_fetch(number = NULL, target = c("source", "primary"))

pr_push()

pr_pull()
```

```
pr_merge_main()

pr_view(number = NULL, target = c("source", "primary"))

pr_pause()

pr_finish(number = NULL, target = c("source", "primary"))

pr_forget()
```

Arguments

| | |
|--------|---|
| branch | Name of a new or existing local branch. If creating a new branch, note this should usually consist of lower case letters, numbers, and -. |
| number | Number of PR. |
| target | Which repo to target? This is only a question in the case of a fork. In a fork, there is some slim chance that you want to consider pull requests against your fork (the primary repo, i.e. origin) instead of those against the source repo (i.e. upstream, which is the default). |

Git/GitHub Authentication

Many usethis functions, including those documented here, potentially interact with GitHub in two different ways:

- Via the GitHub REST API. Examples: create a repo, a fork, or a pull request.
- As a conventional Git remote. Examples: clone, fetch, or push.

Therefore two types of auth can happen and your credentials must be discoverable. Which credentials do we mean?

- A GitHub personal access token (PAT) must be discoverable by the gh package, which is used for GitHub operations via the REST API. See [gh_token_help\(\)](#) for more about getting and configuring a PAT.
- If you use the HTTPS protocol for Git remotes, your PAT is also used for Git operations, such as `git push`. Usethis uses the gert package for this, so the PAT must be discoverable by gert. Generally gert and gh will discover and use the same PAT. This ability to "kill two birds with one stone" is why HTTPS + PAT is our recommended auth strategy for those new to Git and GitHub and PRs.
- If you use SSH remotes, your SSH keys must also be discoverable, in addition to your PAT. The public key must be added to your GitHub account.

Git/GitHub credential management is covered in a dedicated article: [Managing Git\(Hub\) Credentials](#)

For contributors

To contribute to a package, first use `create_from_github("OWNER/REPO")`. This forks the source repository and checks out a local copy.

Next use `pr_init()` to create a branch for your PR. It is best practice to never make commits to the default branch `branch` of a fork (usually named `main` or `master`), because you do not own it. A pull request should always come from a feature branch. It will be much easier to pull upstream changes from the fork parent if you only allow yourself to work in feature branches. It is also much easier for a maintainer to explore and extend your PR if you create a feature branch.

Work locally, in your branch, making changes to files, and committing your work. Once you're ready to create the PR, run `pr_push()` to push your local branch to GitHub, and open a webpage that lets you initiate the PR (or draft PR).

To learn more about the process of making a pull request, read the [Pull Request Helpers](#) vignette.

If you are lucky, your PR will be perfect, and the maintainer will accept it. You can then run `pr_finish()` to delete your PR branch. In most cases, however, the maintainer will ask you to make some changes. Make the changes, then run `pr_push()` to update your PR.

It's also possible that the maintainer will contribute some code to your PR: to get those changes back onto your computer, run `pr_pull()`. It can also happen that other changes have occurred in the package since you first created your PR. You might need to merge the default branch (usually named `main` or `master`) into your PR branch. Do that by running `pr_merge_main()`: this makes sure that your PR is compatible with the primary repo's main line of development. Both `pr_pull()` and `pr_merge_main()` can result in merge conflicts, so be prepared to resolve before continuing.

For maintainers

To download a PR locally so that you can experiment with it, run `pr_fetch()` and select the PR or, if you already know its number, call `pr_fetch(<pr_number>)`. If you make changes, run `pr_push()` to push them back to GitHub. After you have merged the PR, run `pr_finish()` to delete the local branch and remove the remote associated with the contributor's fork.

Overview of all the functions

- `pr_init()`: As a contributor, start work on a new PR by ensuring that your local repo is up-to-date, then creating and checking out a new branch. Nothing is pushed to or created on GitHub until you call `pr_push()`.
- `pr_fetch()`: As a maintainer, review or contribute changes to an existing PR by creating a local branch that tracks the remote PR. `pr_fetch()` does as little work as possible, so you can also use it to resume work on an PR that already has a local branch (where it will also ensure your local branch is up-to-date). If called with no arguments, up to 9 open PRs are offered for interactive selection.
- `pr_resume()`: Resume work on a PR by switching to an existing local branch and pulling any changes from its upstream tracking branch, if it has one. If called with no arguments, up to 9 local branches are offered for interactive selection, with a preference for branches connected to PRs and for branches with recent activity.
- `pr_push()`: The first time it's called, a PR branch is pushed to GitHub and you're taken to a webpage where a new PR (or draft PR) can be created. This also sets up the local branch to track its remote counterpart. Subsequent calls to `pr_push()` make sure the local branch has all the remote changes and, if so, pushes local changes, thereby updating the PR.
- `pr_pull()`: Pulls changes from the local branch's remote tracking branch. If a maintainer has extended your PR, this is how you bring those changes back into your local work.

- `pr_merge_main()`: Pulls changes from the default branch of the source repo into the current local branch. This can be used when the local branch is the default branch or when it's a PR branch.
- `pr_pause()`: Makes sure you're up-to-date with any remote changes in the PR. Then switches back to the default branch and pulls from the source repo. Use `pr_resume()` with name of branch or use `pr_fetch()` to resume using PR number.
- `pr_view()`: Visits the PR associated with the current branch in the browser (default) or the specific PR identified by number. (FYI [browse_github_pulls\(\)](#) is a handy way to visit the list of all PRs for the current project.)
- `pr_forget()`: Does local clean up when the current branch is an actual or notional PR that you want to abandon. Maybe you initiated it yourself, via `pr_init()`, or you used `pr_fetch()` to explore a PR from GitHub. Only does *local* operations: does not update or delete any remote branches, nor does it close any PRs. Alerts the user to any uncommitted or unpushed work that is at risk of being lost. If user chooses to proceed, switches back to the default branch, pulls changes from source repo, and deletes local PR branch. Any associated Git remote is deleted, if the "forgotten" PR was the only branch using it.
- `pr_finish()`: Does post-PR clean up, but does NOT actually merge or close a PR (maintainer should do this in the browser). If number is not given, infers the PR from the upstream tracking branch of the current branch. If number is given, it does not matter whether the PR exists locally. If PR exists locally, alerts the user to uncommitted or unpushed changes, then switches back to the default branch, pulls changes from source repo, and deletes local PR branch. If the PR came from an external fork, any associated Git remote is deleted, provided it's not in use by any other local branches. If the PR has been merged and user has permission, deletes the remote branch (this is the only remote operation that `pr_finish()` potentially does).

Examples

```
## Not run:
pr_fetch(123)

## End(Not run)
```

rename_files

Automatically rename paired R/ and test/ files

Description

- Moves `R/{old}.R` to `R/{new}.R`
- Moves `src/{old}.*` to `src/{new}.*`
- Moves `tests/testthat/test-{old}.R` to `tests/testthat/test-{new}.R`
- Moves `tests/testthat/test-{old}-*.R` to `tests/testthat/test-{new}-*.R` and updates paths in the test file.
- Removes `context()` calls from the test file, which are unnecessary (and discouraged) as of `testthat v2.1.0`.

This is a potentially dangerous operation, so you must be using Git in order to use this function.

Usage

```
rename_files(old, new)
```

Arguments

old, new Old and new file names (with or without .R extensions).

rprofile-helper *Helpers to make useful changes to .Rprofile*

Description

All functions open your .Rprofile and give you the code you need to paste in.

- use_devtools(): makes devtools available in interactive sessions.
- use_usethis(): makes usethis available in interactive sessions.
- use_reprex(): makes reprex available in interactive sessions.
- use_conflicted(): makes conflicted available in interactive sessions.
- use_partial_warnings(): warns on partial matches.

Usage

```
use_conflicted()
```

```
use_reprex()
```

```
use_usethis()
```

```
use_devtools()
```

```
use_partial_warnings()
```

ui_silence *Suppress usethis's messaging*

Description

Execute a bit of code without usethis's normal messaging.

Usage

```
ui_silence(code)
```

Arguments

code Code to execute with usual UI output silenced.

Value

Whatever code returns.

Examples

```
# compare the messaging you see from this:
browse_github("usethis")
# vs. this:
ui_silence(
  browse_github("usethis")
)
```

usethis_options *Options consulted by usethis*

Description

User-configurable options consulted by usethis, which provide a mechanism for setting default behaviors for various functions.

If the built-in defaults don't suit you, set one or more of these options. Typically, this is done in the .Rprofile startup file, which you can open for editing with [edit_r_profile\(\)](#) - this will set the specified options for all future R sessions. Your code will look something like:

```
options(
  usethis.description = list(
    "Authors@R" = utils::person(
      "Jane", "Doe",
      email = "jane@example.com",
      role = c("aut", "cre"),
      comment = c(ORCID = "YOUR-ORCID-ID")
    ),
    License = "MIT + file LICENSE"
  ),
  usethis.destdir = "/path/to/folder/", # for use_course(), create_from_github()
  usethis.protocol = "ssh", # Use ssh git protocol
  usethis.overwrite = TRUE # overwrite files in Git repos without confirmation
)
```

Options for the usethis package

- `usethis.description`: customize the default content of new DESCRIPTION files by setting this option to a named list. If you are a frequent package developer, it is worthwhile to pre-configure your preferred name, email, license, etc. See the example above and the [article on usethis setup](#) for more details.
- `usethis.destdir`: Default directory in which to place new projects downloaded by `use_course()` and `create_from_github()`. If this option is unset, the user's Desktop or similarly conspicuous place will be used.
- `usethis.protocol`: specifies your preferred transport protocol for Git. Either "https" (default) or "ssh":
 - `usethis.protocol = "https"` implies `https://github.com/<OWNER>/<REPO>.git`
 - `usethis.protocol = "ssh"` implies `git@github.com:<OWNER>/<REPO>.git`

You can also change this for the duration of your R session with `use_git_protocol()`.

- `usethis.overwrite`: If TRUE, usethis overwrites an existing file without asking for user confirmation if the file is inside a Git repo. The rationale is that the normal Git workflow makes it easy to see and selectively accept/discard any proposed changes.
- `usethis.quiet`: Set to TRUE to suppress user-facing messages. Default FALSE.
- `usethis.allow_nested_project`: Whether or not to allow you to create a project inside another project. This is rarely a good idea, so this option defaults to FALSE.

use_addin

Add minimal RStudio Addin binding

Description

This function helps you add a minimal **RStudio Addin** binding to `inst/rstudio/addins.dcf`.

Usage

```
use_addin(addin = "new_addin", open = rlang::is_interactive())
```

Arguments

| | |
|--------------------|---|
| <code>addin</code> | Name of the addin function, which should be defined in the R folder. |
| <code>open</code> | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |

use_air

*Configure a project to use Air***Description**

Air is an extremely fast R code formatter. This function sets up a project to use Air. Specifically, it:

- Creates an empty `air.toml` configuration file. If either an `air.toml` or `.air.toml` file already existed, nothing is changed. If the project is an R package, `.Rbuildignore` is updated to ignore this file.
- Creates a `.vscode/` directory and adds recommended settings to `.vscode/settings.json` and `.vscode/extensions.json`. These settings are used by the Air extension installed through either VS Code or Positron, see the Installation section for more details. Specifically it:
 - Sets `editor.formatOnSave = true` for R and Quarto files to enable formatting on every save.
 - Sets `editor.defaultFormatter` to Air for R files to ensure that Air is always selected as the formatter for this project. Likewise, sets the default formatter for Quarto.
 - Sets the Air extension as a "recommended" extension for this project, which triggers a notification for contributors coming to this project that don't yet have the Air extension installed.

If the project is an R package, `.Rbuildignore` is updated to ignore the `.vscode/` directory.

If you'd like to opt out of VS Code / Positron specific setup, set `vscode = FALSE`, but remember that even if you work in RStudio, other contributors may prefer another editor.

Note that "using Air" breaks down into a few steps, and `use_air()` does *one* of them. Here's an overview:

- Installation: Air might already be included in your IDE (e.g. Positron) or can be added as an external formatter (e.g. RStudio) or as an extension (e.g. VS Code). Read the guide that applies to your situation:
 - [Air in an editor](#)
 - [Air at the command line](#)
- Configuration: `use_air()` does this!
- Invocation: There are many ways to run Air. In an IDE, you can expect support for moves like "format on save", "format selection", and so on. At the command line, you can format individual files or entire directories.
- Continuous integration: Two workflows are available for running Air via GitHub Actions: `format-suggest` or `format-check`. Learn more in [Air's documentation of its GHA integrations](#). You can set up either workflow in your project like so:

```
use_github_action(url = "https://github.com/posit-dev/setup-air/blob/main/examples/format-suggest")
use_github_action(url = "https://github.com/posit-dev/setup-air/blob/main/examples/format-check")
```

Usage

```
use_air(vscode = TRUE)
```

Arguments

- vscode Either:
- TRUE to set up VS Code and Positron specific Air settings. This is the default.
 - FALSE to opt out of those settings.

Examples

```
## Not run:
# Prepare an R package or project to use Air
use_air()

## End(Not run)
```

| | |
|------------|--|
| use_author | <i>Add an author to the Authors@R field in DESCRIPTION</i> |
|------------|--|

Description

use_author() adds a person to the Authors@R field of the DESCRIPTION file, creating that field if necessary. It will not modify, e.g., the role(s) or email of an existing author (judged using their "Given Family" name). For that we recommend editing DESCRIPTION directly. Or, for programmatic use, consider calling the more specialized functions available in the **desc** package directly.

use_author() also surfaces two other situations you might want to address:

- Explicit use of the fields Author or Maintainer. We recommend switching to the more modern Authors@R field instead, because it offers richer metadata for various downstream uses. (Note that Authors@R is *eventually* processed to create Author and Maintainer fields, but only when the tar.gz is built from package source.)
- Presence of the fake author placed by [create_package\(\)](#) and [use_description\(\)](#). This happens when **usethis** has to create a DESCRIPTION file and the user hasn't given any author information via the fields argument or the global option "usethis.description". The placeholder looks something like First Last <first.last@example.com> [aut, cre] and use_author() offers to remove it in interactive sessions.

Usage

```
use_author(given = NULL, family = NULL, ..., role = "ctb")
```

Arguments

- given a character vector with the *given* names, or a list thereof.
- family a character string with the *family* name, or a list thereof.
- ... Arguments passed on to [utils::person](#)
- middle a character string with the collapsed middle name(s). Deprecated, see **Details**.

`email` a character string (or vector) giving an e-mail address (each), or a list thereof.

`comment` a character string (or vector) providing comments, or a list thereof.

`first` a character string giving the first name. Deprecated, see **Details**.

`last` a character string giving the last name. Deprecated, see **Details**.

`role` a character vector specifying the role(s) of the person (see **Details**), or a list thereof.

Examples

```
## Not run:
use_author(
  given = "Lucy",
  family = "van Pelt",
  role = c("aut", "cre"),
  email = "lucy@example.com",
  comment = c(ORCID = "LUCY-ORCID-ID")
)

use_author("Charlie", "Brown")

## End(Not run)
```

use_blank_slate

Don't save/load user workspace between sessions

Description

R can save and reload the user's workspace between sessions via an `.RData` file in the current directory. However, long-term reproducibility is enhanced when you turn this feature off and clear R's memory at every restart. Starting with a blank slate provides timely feedback that encourages the development of scripts that are complete and self-contained. More detail can be found in the blog post [Project-oriented workflow](#).

Usage

```
use_blank_slate(scope = c("user", "project"))
```

Arguments

`scope` Edit globally for the current **user**, or locally for the current **project**

| | |
|------------------|-----------------------------------|
| use_build_ignore | <i>Add files to .Rbuildignore</i> |
|------------------|-----------------------------------|

Description

.Rbuildignore has a regular expression on each line, but it's usually easier to work with specific file names. By default, use_build_ignore() will (crudely) turn a filename into a regular expression that will only match that path. Repeated entries will be silently removed.

use_build_ignore() is designed to ignore *individual* files. If you want to ignore *all* files with a given extension, consider providing an "as-is" regular expression, using escape = FALSE; see examples.

Usage

```
use_build_ignore(files, escape = TRUE)
```

Arguments

| | |
|--------|---|
| files | Character vector of path names. |
| escape | If TRUE, the default, will escape . to \. and surround with ^ and \$. |

Examples

```
## Not run:
# ignore all Excel files
use_build_ignore("[.]xlsx$", escape = FALSE)

## End(Not run)
```

| | |
|--------------|-----------------------------------|
| use_citation | <i>Create a CITATION template</i> |
|--------------|-----------------------------------|

Description

Use this if you want to encourage users of your package to cite an article or book.

Usage

```
use_citation()
```

use_code_of_conduct *Add a code of conduct*

Description

Adds a CODE_OF_CONDUCT.md file to the active project and lists in .Rbuildignore, in the case of a package. The goal of a code of conduct is to foster an environment of inclusiveness, and to explicitly discourage inappropriate behaviour. The template comes from <https://www.contributor-covenant.org>, version 2.1: https://www.contributor-covenant.org/version/2/1/code_of_conduct/.

Usage

```
use_code_of_conduct(contact, path = NULL)
```

Arguments

| | |
|---------|---|
| contact | Contact details for making a code of conduct report. Usually an email address. |
| path | Path of the directory to put CODE_OF_CONDUCT.md in, relative to the active project. Passed along to <code>use_directory()</code> . Default is to locate at top-level, but <code>.github/</code> is also common. |

Details

If your package is going to CRAN, the link to the CoC in your README must be an absolute link to a rendered website as CODE_OF_CONDUCT.md is not included in the package sent to CRAN. `use_code_of_conduct()` will automatically generate this link if (1) you use `pkgdown` and (2) have set the `url` field in `_pkgdown.yml`; otherwise it will link to a copy of the CoC on <https://www.contributor-covenant.org>.

use_coverage *Test coverage*

Description

Adds test coverage reporting to a package, using either Codecov (<https://codecov.io>) or Coveralls (<https://coveralls.io>).

Usage

```
use_coverage(type = c("codecov", "coveralls"), repo_spec = NULL)

use_covr_ignore(files)
```

Arguments

| | |
|-----------|--|
| type | Which web service to use. |
| repo_spec | Optional GitHub repo specification in this form: owner/repo. This can usually be inferred from the GitHub remotes of active project. |
| files | Character vector of file globs. |

| | |
|-----------|--------------------------------------|
| use_cpp11 | <i>Use C++ via the cpp11 package</i> |
|-----------|--------------------------------------|

Description

Adds infrastructure needed to use the `cpp11` package, a header-only R package that helps R package developers handle R objects with C++ code:

- Creates `src/`
- Adds `cpp11` to DESCRIPTION
- Creates `src/code.cpp`, an initial placeholder `.cpp` file

Usage

```
use_cpp11()
```

| | |
|-------------------|---------------------------------|
| use_cran_comments | <i>CRAN submission comments</i> |
|-------------------|---------------------------------|

Description

Creates `cran-comments.md`, a template for your communications with CRAN when submitting a package. The goal is to clearly communicate the steps you have taken to check your package on a wide range of operating systems. If you are submitting an update to a package that is used by other packages, you also need to summarize the results of your [reverse dependency checks](#).

Usage

```
use_cran_comments(open = rlang::is_interactive())
```

Arguments

| | |
|------|---|
| open | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |
|------|---|

use_data *Create package data*

Description

`use_data()` makes it easy to save package data in the correct format. I recommend you save scripts that generate package data in `data-raw`: use `use_data_raw()` to set it up. You also need to document exported datasets.

Usage

```
use_data(
  ...,
  internal = FALSE,
  overwrite = FALSE,
  compress = "bzip2",
  version = 3,
  ascii = FALSE
)

use_data_raw(name = "DATASET", open = rlang::is_interactive())
```

Arguments

| | |
|------------------------|---|
| <code>...</code> | Unquoted names of existing objects to save. |
| <code>internal</code> | If <code>FALSE</code> , saves each object in its own <code>.rda</code> file in the <code>data/</code> directory. These data files bypass the usual export mechanism and are available whenever the package is loaded (or via <code>data()</code> if <code>LazyData</code> is not true). If <code>TRUE</code> , stores all objects in a single <code>R/sysdata.rda</code> file. Objects in this file follow the usual export rules. Note that this means they will be exported if you are using the common <code>exportPattern()</code> rule which exports all objects except for those that start with <code>..</code> |
| <code>overwrite</code> | By default, <code>use_data()</code> will not overwrite existing files. If you really want to do so, set this to <code>TRUE</code> . |
| <code>compress</code> | Choose the type of compression used by <code>save()</code> . Should be one of <code>"gzip"</code> , <code>"bzip2"</code> , or <code>"xz"</code> . |
| <code>version</code> | The serialization format version to use. The default, 3, can only be read by R versions 3.5.0 and higher. For R 1.4.0 to 3.5.3, use version 2. |
| <code>ascii</code> | if <code>TRUE</code> , an ASCII representation of the data is written. The default value of <code>ascii</code> is <code>FALSE</code> which leads to a binary file being written. If <code>NA</code> and <code>version >= 2</code> , a different ASCII representation is used which writes double/complex numbers as binary fractions. |
| <code>name</code> | Name of the dataset to be prepared for inclusion in the package. |
| <code>open</code> | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |

See Also

The [data chapter](#) of **R Packages**.

Examples

```
## Not run:
x <- 1:10
y <- 1:100

use_data(x, y) # For external use
use_data(x, y, internal = TRUE) # For internal use

## End(Not run)
## Not run:
use_data_raw("daisy")

## End(Not run)
```

| | |
|----------------|---|
| use_data_table | <i>Prepare for importing data.table</i> |
|----------------|---|

Description

use_data_table() imports the data.table() function from the data.table package, as well as several important symbols: :=, .SD, .BY, .N, .I, .GRP, .NGRP, .EACHI. This is a minimal setup and you can learn much more in the "Importing data.table" vignette: <https://rdatatable.gitlab.io/data.table/articles/data-table/>. In addition to importing these functions, use_data_table() also blocks the usage of data.table in the Depends field of the DESCRIPTION file; data.table should be used as an *imported* or *suggested* package only. See this [discussion](#).

Usage

```
use_data_table()
```

| | |
|-----------------|--|
| use_description | <i>Create or modify a DESCRIPTION file</i> |
|-----------------|--|

Description

use_description() creates a DESCRIPTION file. Although mostly associated with R packages, a DESCRIPTION file can also be used to declare dependencies for a non-package project. Within such a project, devtools::install_deps() can then be used to install all the required packages. Note that, by default, use_description() checks for a CRAN-compliant package name. You can turn this off with check_name = FALSE.

usethis consults the following sources, in this order, to set DESCRIPTION fields:

- fields argument of `create_package()` or `use_description()`
- `getOption("usethis.description")`
- Defaults built into `usethis`

The fields discovered via options or the `usethis` package can be viewed with `use_description_defaults()`.

If you create a lot of packages, consider storing personalized defaults as a named list in an option named `"usethis.description"`. Here's an example of code to include in `.Rprofile`, which can be opened via `edit_r_profile()`:

```
options(
  usethis.description = list(
    "Authors@R" = utils::person(
      "Jane", "Doe",
      email = "jane@example.com",
      role = c("aut", "cre"),
      comment = c(ORCID = "YOUR-ORCID-ID")
    ),
    Language = "es",
    License = "MIT + file LICENSE"
  )
)
```

Prior to `usethis` v2.0.0, `getOption("devtools.desc")` was consulted for backwards compatibility, but now only the `"usethis.description"` option is supported.

Usage

```
use_description(fields = list(), check_name = TRUE, roxygen = TRUE)
```

```
use_description_defaults(package = NULL, roxygen = TRUE, fields = list())
```

Arguments

| | |
|-------------------------|--|
| <code>fields</code> | A named list of fields to add to DESCRIPTION, potentially overriding default values. Default values are taken from the <code>"usethis.description"</code> option or the <code>usethis</code> package (in that order), and can be viewed with <code>use_description_defaults()</code> . |
| <code>check_name</code> | Whether to check if the name is valid for CRAN and throw an error if not. |
| <code>roxygen</code> | If TRUE, sets <code>RoxygenNote</code> to current <code>roxygen2</code> version |
| <code>package</code> | Package name |

See Also

The [description chapter](#) of [R Packages](#)

Examples

```
## Not run:
use_description()

use_description(fields = list(Language = "es"))

use_description_defaults()

## End(Not run)
```

| | |
|---------------|------------------------|
| use_directory | <i>Use a directory</i> |
|---------------|------------------------|

Description

use_directory() creates a directory (if it does not already exist) in the project's top-level directory. This function powers many of the other use_ functions such as [use_data\(\)](#) and [use_vignette\(\)](#).

Usage

```
use_directory(path, ignore = FALSE)
```

Arguments

| | |
|--------|---|
| path | Path of the directory to create, relative to the project. |
| ignore | Should the newly created file be added to .Rbuildignore? |

Examples

```
## Not run:
use_directory("inst")

## End(Not run)
```

| | |
|---------|------------------------------------|
| use_git | <i>Initialise a git repository</i> |
|---------|------------------------------------|

Description

use_git() initialises a Git repository and adds important files to .gitignore. If user consents, it also makes an initial commit.

Usage

```
use_git(message = "Initial commit")
```

Arguments

message Message to use for first commit.

See Also

Other git helpers: [use_git_config\(\)](#), [use_git_hook\(\)](#), [use_git_ignore\(\)](#)

Examples

```
## Not run:  
use_git()  
  
## End(Not run)
```

| | |
|------------|---|
| use_github | <i>Connect a local repo with GitHub</i> |
|------------|---|

Description

use_github() takes a local project and:

- Checks that the initial state is good to go:
 - Project is already a Git repo
 - Current branch is the default branch, e.g. main or master
 - No uncommitted changes
 - No pre-existing origin remote
- Creates an associated repo on GitHub
- Adds that GitHub repo to your local repo as the origin remote
- Makes an initial push to GitHub
- Calls [use_github_links\(\)](#), if the project is an R package
- Configures origin/DEFAULT to be the upstream branch of the local DEFAULT branch, e.g. main or master

See below for the authentication setup that is necessary for all of this to work.

Usage

```
use_github(  
  organisation = NULL,  
  private = FALSE,  
  visibility = c("public", "private", "internal"),  
  protocol = git_protocol(),  
  host = NULL  
)
```

Arguments

| | |
|--------------|---|
| organisation | If supplied, the repo will be created under this organisation, instead of the login associated with the GitHub token discovered for this host. The user's role and the token's scopes must be such that you have permission to create repositories in this organisation. |
| private | If TRUE, creates a private repository. |
| visibility | Only relevant for organisation-owned repos associated with certain GitHub Enterprise products. The special "internal" visibility grants read permission to all organisation members, i.e. it's intermediate between "private" and "public", within GHE. When specified, visibility takes precedence over private = TRUE/FALSE. |
| protocol | One of "https" or "ssh" |
| host | GitHub host to target, passed to the <code>.api_url</code> argument of <code>gh::gh()</code> . If unspecified, <code>gh</code> defaults to "https://api.github.com", although <code>gh</code> 's default can be customised by setting the <code>GITHUB_API_URL</code> environment variable. For a hypothetical GitHub Enterprise instance, either "https://github.acme.com/api/v3" or "https://github.acme.com" is acceptable. |

Git/GitHub Authentication

Many `usethis` functions, including those documented here, potentially interact with GitHub in two different ways:

- Via the GitHub REST API. Examples: create a repo, a fork, or a pull request.
- As a conventional Git remote. Examples: clone, fetch, or push.

Therefore two types of auth can happen and your credentials must be discoverable. Which credentials do we mean?

- A GitHub personal access token (PAT) must be discoverable by the `gh` package, which is used for GitHub operations via the REST API. See `gh_token_help()` for more about getting and configuring a PAT.
- If you use the HTTPS protocol for Git remotes, your PAT is also used for Git operations, such as `git push`. `usethis` uses the `gert` package for this, so the PAT must be discoverable by `gert`. Generally `gert` and `gh` will discover and use the same PAT. This ability to "kill two birds with one stone" is why HTTPS + PAT is our recommended auth strategy for those new to Git and GitHub and PRs.
- If you use SSH remotes, your SSH keys must also be discoverable, in addition to your PAT. The public key must be added to your GitHub account.

Git/GitHub credential management is covered in a dedicated article: [Managing Git\(Hub\) Credentials](#)

Examples

```
## Not run:
pkgpath <- file.path(tempdir(), "testpkg")
create_package(pkgpath)
```

```
## now, working inside "testpkg", initialize git repository
use_git()

## create github repository and configure as git remote
use_github()

## End(Not run)
```

use_github_action *Set up a GitHub Actions workflow*

Description

Sets up continuous integration (CI) for an R package that is developed on GitHub using **GitHub Actions** (GHA). CI can be used to trigger various operations for each push or pull request, e.g. running R CMD check or building and deploying a pkgdown site.

Core workflows:

There are three particularly important workflows that are used by many packages:

- `check-standard`: Run R CMD check using R-latest on Linux, Mac, and Windows, and using R-devel and R-oldrel on Linux. This is a good baseline if you plan on submitting your package to CRAN.
- `test-coverage`: Compute test coverage and report to <https://about.codecov.io> by calling `covr::codecov()`.
- `pkgdown`: Automatically build and publish a pkgdown website. But we recommend instead calling `use_pkgdown_github_pages()`, which sets up the pkgdown workflow AND performs other important set up.

If you call `use_github_action()` without arguments, you'll get a choice of some recommended workflows. Otherwise you can specify the name of any workflow provided by `r-lib/actions`, which are listed at <https://github.com/r-lib/actions/tree/v2/examples>. Finally you can supply the full url to any workflow on GitHub.

Other workflows:

Other specific workflows are worth mentioning:

- `format-suggest` or `format-check` from **Air**: **[Experimental]** Either of these workflows is a great way to keep your code well-formatted once you adopt Air in a project (possibly via `use_air()`). Here's how to set them up:

```
use_github_action(url = "https://github.com/posit-dev/setup-air/blob/main/examples/format-suggest")
use_github_action(url = "https://github.com/posit-dev/setup-air/blob/main/examples/format-check")
```

Learn more from [Air's documentation of its GHA integrations](#).

- `pr-commands`: **[Superseded]** Enables the use of two R-specific commands in pull request issue comments: `/document` to run `roxygen2::roxygenise()` and `/style` to run `styler::style_pkg()`. Both will update the PR with any changes once they're done.

We don't recommend new adoption of the `pr-commands` workflow. For code formatting, the Air workflows described above are preferred. We plan to re-implement documentation updates using a similar approach.

Usage

```
use_github_action(
  name = NULL,
  ref = NULL,
  url = NULL,
  save_as = NULL,
  readme = NULL,
  ignore = TRUE,
  open = FALSE,
  badge = NULL
)
```

Arguments

| | |
|---------|---|
| name | Name of one of the example workflows from https://github.com/r-lib/actions/tree/v2/examples (with or without extension), e.g. "pkgdown", "check-standard.yaml". If the name starts with check-, save_as defaults to R-CMD-check.yaml and badge defaults to TRUE. |
| ref | Desired Git reference, usually the name of a tag ("v2") or branch ("main"). Other possibilities include a commit SHA ("d1c516d") or "HEAD" (meaning "tip of remote's default branch"). If not specified, defaults to the latest published release of r-lib/actions (https://github.com/r-lib/actions/releases). |
| url | The full URL to a .yaml file on GitHub. See more details in use_github_file() . |
| save_as | Name of the local workflow file. Defaults to name or fs::path_file(url). Do not specify any other part of the path; the parent directory will always be .github/workflows, within the active project. |
| readme | The full URL to a README file that provides more details about the workflow. Ignored when url is NULL. |
| ignore | Should the newly created file be added to .Rbuildignore? |
| open | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |
| badge | Should we add a badge to the README? |

Examples

```
## Not run:
use_github_action()

use_github_action("check-standard")

use_github_action("pkgdown")

use_github_action(
  url = "https://github.com/posit-dev/setup-air/blob/main/examples/format-suggest.yaml"
)

## End(Not run)
```

use_github_file *Copy a file from any GitHub repo into the current project*

Description

Gets the content of a file from GitHub, from any repo the user can read, and writes it into the active project. This function wraps an endpoint of the GitHub API which supports specifying a target reference (i.e. branch, tag, or commit) and which follows symlinks.

Usage

```
use_github_file(
  repo_spec,
  path = NULL,
  save_as = NULL,
  ref = NULL,
  ignore = FALSE,
  open = FALSE,
  overwrite = FALSE,
  host = NULL
)
```

Arguments

| | |
|-----------|--|
| repo_spec | A string identifying the GitHub repo or, alternatively, a GitHub file URL. Acceptable forms: <ul style="list-style-type: none"> • Plain OWNER/REPO spec • A blob URL, such as "https://github.com/OWNER/REPO/blob/REF/path/to/some/file" • A raw URL, such as "https://raw.githubusercontent.com/OWNER/REPO/REF/path/to/some/f" <p>In the case of a URL, the path, ref, and host are extracted from it, in addition to the repo_spec.</p> |
| path | Path of file to copy, relative to the GitHub repo it lives in. This is extracted from repo_spec when user provides a URL. |
| save_as | Path of file to create, relative to root of active project. Defaults to the last part of path, in the sense of <code>basename(path)</code> or <code>fs::path_file(path)</code> . |
| ref | The name of a branch, tag, or commit. By default, the file at path will be copied from its current state in the repo's default branch. This is extracted from repo_spec when user provides a URL. |
| ignore | Should the newly created file be added to <code>.Rbuildignore</code> ? |
| open | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |
| overwrite | Force overwrite of existing file? |

`host` GitHub host to target, passed to the `.api_url` argument of `gh::gh()`. If unspecified, `gh` defaults to "https://api.github.com", although `gh`'s default can be customised by setting the `GITHUB_API_URL` environment variable.

For a hypothetical GitHub Enterprise instance, either "https://github.acme.com/api/v3" or "https://github.acme.com" is acceptable.

Value

A logical indicator of whether a file was written, invisibly.

Examples

```
## Not run:
use_github_file(
  "https://github.com/r-lib/actions/blob/v2/examples/check-standard.yaml"
)

use_github_file(
  "r-lib/actions",
  path = "examples/check-standard.yaml",
  ref = "v2",
  save_as = ".github/workflows/R-CMD-check.yaml"
)

## End(Not run)
```

use_github_labels *Manage GitHub issue labels*

Description

`use_github_labels()` can create new labels, update colours and descriptions, and optionally delete GitHub's default labels (if `delete_default = TRUE`). It will never delete labels that have associated issues.

`use_tidy_github_labels()` calls `use_github_labels()` with tidyverse conventions powered by `tidy_labels()`, `tidy_labels_rename()`, `tidy_label_colours()` and `tidy_label_descriptions()`.

tidyverse label usage:

Labels are used as part of the issue-triage process, designed to minimise the time spent re-reading issues. The absence of a label indicates that an issue is new, and has yet to be triaged.

There are four mutually exclusive labels that indicate the overall "type" of issue:

- `bug`: an unexpected problem or unintended behavior.
- `documentation`: requires changes to the docs.
- `feature`: feature requests and enhancement.
- `upkeep`: general package maintenance work that makes future development easier.

Then there are five labels that are needed in most repositories:

- `breaking change`: issue/PR will requires a breaking change so should be not be included in patch releases.
- `reprex` indicates that an issue does not have a minimal reproducible example, and that a reply has been sent requesting one from the user.
- `good first issue` indicates a good issue for first-time contributors.
- `help wanted` indicates that a maintainer wants help on an issue.
- `wip` indicates that someone is working on it or has promised to.

Finally most larger repos will accumulate their own labels for specific areas of functionality. For example, `usethis` has labels like "description", "paths", "readme", because time has shown these to be common sources of problems. These labels are helpful for grouping issues so that you can tackle related problems at the same time.

Repo-specific issues should have a grey background (`#eeeeee`) and an emoji. This keeps the issue page visually harmonious while still giving enough variation to easily distinguish different types of label.

Usage

```
use_github_labels(
  labels = character(),
  rename = character(),
  colours = character(),
  descriptions = character(),
  delete_default = FALSE
)

use_tidy_github_labels()

tidy_labels()

tidy_labels_rename()

tidy_label_colours()

tidy_label_descriptions()
```

Arguments

| | |
|------------------------------------|---|
| <code>labels</code> | A character vector giving labels to add. |
| <code>rename</code> | A named vector with names giving old names and values giving new names. |
| <code>colours, descriptions</code> | Named character vectors giving hexadecimal colours (like <code>e02a2a</code>) and longer descriptions. The names should match label names, and anything unmatched will be left unchanged. If you create a new label, and don't supply colours, it will be given a random colour. |
| <code>delete_default</code> | If TRUE, removes GitHub default labels that do not appear in the <code>labels</code> vector and that do not have associated issues. |

Examples

```
## Not run:
# typical use in, e.g., a new tidyverse project
use_github_labels(delete_default = TRUE)

# create labels without changing colours/descriptions
use_github_labels(
  labels = c("foofy", "foofier", "foofiest"),
  colours = NULL,
  descriptions = NULL
)

# change descriptions without changing names/colours
use_github_labels(
  labels = NULL,
  colours = NULL,
  descriptions = c("foofiest" = "the foofiest issue you ever saw")
)

## End(Not run)
```

use_github_links

Use GitHub links in URL and BugReports

Description

Populates the URL and BugReports fields of a GitHub-using R package with appropriate links. The GitHub repo to link to is determined from the current project's GitHub remotes:

- If we are not working with a fork, this function expects origin to be a GitHub remote and the links target that repo.
- If we are working in a fork, this function expects to find two GitHub remotes: origin (the fork) and upstream (the fork's parent) remote. In an interactive session, the user can confirm which repo to use for the links. In a noninteractive session, links are formed using upstream.

Usage

```
use_github_links(overwrite = FALSE)
```

Arguments

overwrite By default, use_github_links() will not overwrite existing fields. Set to TRUE to overwrite existing links.

Examples

```
## Not run:
use_github_links()

## End(Not run)
```

use_github_pages *Configure a GitHub Pages site*

Description

Activates or reconfigures a GitHub Pages site for a project hosted on GitHub. This function anticipates two specific usage modes:

- Publish from the root directory of a gh-pages branch, which is assumed to be only (or at least primarily) a remote branch. Typically the gh-pages branch is managed by an automatic "build and deploy" job, such as the one configured by `use_github_action("pkgdown")`.
- Publish from the "/docs" directory of a "regular" branch, probably the repo's default branch. The user is assumed to have a plan for how they will manage the content below "/docs".

Usage

```
use_github_pages(branch = "gh-pages", path = "/", cname = NA)
```

Arguments

| | |
|--------------|--|
| branch, path | <p>Branch and path for the site source. The default of branch = "gh-pages" and path = "/" reflects strong GitHub support for this configuration: when a gh-pages branch is first created, it is <i>automatically</i> published to Pages, using the source found in "/". If a gh-pages branch does not yet exist on the host, <code>use_github_pages()</code> creates an empty, orphan remote branch.</p> <p>The most common alternative is to use the repo's default branch, coupled with path = "/docs". It is the user's responsibility to ensure that this branch pre-exists on the host.</p> <p>Note that GitHub does not support an arbitrary path and, at the time of writing, only "/" or "/docs" are accepted.</p> |
| cname | <p>Optional, custom domain name. The NA default means "don't set or change this", whereas a value of NULL removes any previously configured custom domain.</p> <p>Note that this <i>can</i> add or modify a CNAME file in your repository. If you are using Pages to host a pkgdown site, it is better to specify its URL in the pkgdown config file and let pkgdown manage CNAME.</p> |

Value

Site metadata returned by the GitHub API, invisibly

See Also

- `use_pkgdown_github_pages()` combines `use_github_pages()` with other functions to fully configure a pkgdown site
- <https://docs.github.com/en/pages>
- <https://docs.github.com/en/rest/pages>

Examples

```
## Not run:
use_github_pages()
use_github_pages(branch = git_default_branch(), path = "/docs")

## End(Not run)
```

```
use_github_release      Publish a GitHub release
```

Description

Pushes the current branch (if safe) then publishes a GitHub release for the latest CRAN submission. If you use `devtools::submit_cran()` to submit to CRAN, information about the submitted state is captured in a CRAN-SUBMISSION file. `use_github_release()` uses this info to populate the GitHub release notes and, after success, deletes the file. In the absence of such a file, we assume that current state (SHA of HEAD, package version, NEWS) is the submitted state.

Usage

```
use_github_release(publish = TRUE)
```

Arguments

```
publish      If TRUE, publishes a release. If FALSE, creates a draft release.
```

```
use_gitlab_ci      Continuous integration setup and badges
```

Description**[Questioning]**

These functions are not actively used by the tidyverse team, and may not currently work. Use at your own risk.

Sets up third-party continuous integration (CI) services for an R package on GitLab or CircleCI. These functions:

- Add service-specific configuration files and add them to `.Rbuildignore`.
- Activate a service or give the user a detailed prompt.
- Provide the markdown to insert a badge into README.

Usage

```
use_gitlab_ci()

use_circleci(browse = rlang::is_interactive(), image = "rocker/verse:latest")

use_circleci_badge(repo_spec = NULL)
```

Arguments

| | |
|-----------|---|
| browse | Open a browser window to enable automatic builds for the package. |
| image | The Docker image to use for build. Must be available on DockerHub . The rocker/verse image includes TeXLive, pandoc, and the tidyverse packages. For a minimal image, try rocker/r-ver . To specify a version of R, change the tag from latest to the version you want, e.g. rocker/r-ver:3.5.3. |
| repo_spec | Optional GitHub repo specification in this form: owner/repo. This can usually be inferred from the GitHub remotes of active project. |

use_gitlab_ci()

Adds a basic `.gitlab-ci.yml` to the top-level directory of a package. This is a configuration file for the **GitLab CI/CD** continuous integration service.

use_circleci()

Adds a basic `.circleci/config.yml` to the top-level directory of a package. This is a configuration file for the **CircleCI** continuous integration service.

use_circleci_badge()

Only adds the **Circle CI** badge. Use for a project where Circle CI is already configured.

| | |
|----------------|----------------------|
| use_git_config | <i>Configure Git</i> |
|----------------|----------------------|

Description

Sets Git options, for either the user or the project ("global" or "local", in Git terminology). Wraps `gert::git_config_set()` and `gert::git_config_global_set()`. To inspect Git config, see `gert::git_config()`.

Usage

```
use_git_config(scope = c("user", "project"), ...)
```

Arguments

| | |
|-------|---|
| scope | Edit globally for the current user , or locally for the current project |
| ... | Name-value pairs, processed as <code><dynamic-dots></code> . |

Value

Invisibly, the previous values of the modified components, as a named list.

See Also

Other git helpers: [use_git\(\)](#), [use_git_hook\(\)](#), [use_git_ignore\(\)](#)

Examples

```
## Not run:
# set the user's global user.name and user.email
use_git_config(user.name = "Jane", user.email = "jane@example.org")

# set the user.name and user.email locally, i.e. for current repo/project
use_git_config(
  scope = "project",
  user.name = "Jane",
  user.email = "jane@example.org"
)

## End(Not run)
```

use_git_hook

Add a git hook

Description

Sets up a git hook using the specified script. Creates a hook directory if needed, and sets correct permissions on hook.

Usage

```
use_git_hook(hook, script)
```

Arguments

| | |
|--------|---|
| hook | Hook name. One of "pre-commit", "prepare-commit-msg", "commit-msg", "post-commit", "applypatch-msg", "pre-applypatch", "post-applypatch", "pre-rebase", "post-rewrite", "post-checkout", "post-merge", "pre-push", "pre-auto-gc". |
| script | Text of script to run |

See Also

Other git helpers: [use_git\(\)](#), [use_git_config\(\)](#), [use_git_ignore\(\)](#)

use_git_ignore *Tell Git to ignore files*

Description

Tell Git to ignore files

Usage

```
use_git_ignore(ignores, directory = ".")
```

Arguments

ignores Character vector of ignores, specified as file globs.
directory Directory relative to active project to set ignores

See Also

Other git helpers: [use_git\(\)](#), [use_git_config\(\)](#), [use_git_hook\(\)](#)

use_git_remote *Configure and report Git remotes*

Description

Two helpers are available:

- `use_git_remote()` sets the remote associated with name to url.
- `git_remotes()` reports the configured remotes, similar to `git remote -v`.

Usage

```
use_git_remote(name = "origin", url, overwrite = FALSE)
```

```
git_remotes()
```

Arguments

name A string giving the short name of a remote.
url A string giving the url of a remote.
overwrite Logical. Controls whether an existing remote can be modified.

Value

Named list of Git remotes.

Examples

```

## Not run:
# see current remotes
git_remotes()

# add new remote named 'foo', a la `git remote add <name> <url>`
use_git_remote(name = "foo", url = "https://github.com/<OWNER>/<REPO>.git")

# remove existing 'foo' remote, a la `git remote remove <name>`
use_git_remote(name = "foo", url = NULL, overwrite = TRUE)

# change URL of remote 'foo', a la `git remote set-url <name> <newurl>`
use_git_remote(
  name = "foo",
  url = "https://github.com/<OWNER>/<REPO>.git",
  overwrite = TRUE
)

# Scenario: Fix remotes when you cloned someone's repo, but you should
# have fork-and-cloned (in order to make a pull request).

# Store origin = main repo's URL, e.g., "git@github.com:<OWNER>/<REPO>.git"
upstream_url <- git_remotes()[["origin"]]

# IN THE BROWSER: fork the main GitHub repo and get your fork's remote URL
my_url <- "git@github.com:<ME>/<REPO>.git"

# Rotate the remotes
use_git_remote(name = "origin", url = my_url)
use_git_remote(name = "upstream", url = upstream_url)
git_remotes()

# Scenario: Add upstream remote to a repo that you fork-and-cloned, so you
# can pull upstream changes.
# Note: If you fork-and-clone via `usethis::create_from_github()`, this is
# done automatically!

# Get URL of main GitHub repo, probably in the browser
upstream_url <- "git@github.com:<OWNER>/<REPO>.git"
use_git_remote(name = "upstream", url = upstream_url)

## End(Not run)

```

use_import_from

Import a function from another package

Description

use_import_from() imports a function from another package by adding the roxygen2 @importFrom tag to the package-level documentation (which can be created with [use_package_doc\(\)](#)). Import-

ing a function from another package allows you to refer to it without a namespace (e.g., `fun()` instead of `package::fun()`).

`use_import_from()` also re-documents the NAMESPACE, and re-load the current package. This ensures that `fun` is immediately available in your development session.

Usage

```
use_import_from(package, fun, load = is_interactive())
```

Arguments

| | |
|---------|--|
| package | Package name |
| fun | A vector of function names |
| load | Logical. Re-load with <code>pkgload::load_all()</code> ? |

Value

Invisibly, TRUE if the package document has changed, FALSE if not.

Examples

```
## Not run:  
use_import_from("glue", "glue")  
  
## End(Not run)
```

use_jenkins

Create Jenkinsfile for Jenkins CI Pipelines

Description

`use_jenkins()` adds a basic Jenkinsfile for R packages to the project root directory. The Jenkinsfile stages take advantage of calls to `make`, and so calling this function will also run `use_make()` if a Makefile does not already exist at the project root.

Usage

```
use_jenkins()
```

See Also

The [documentation on Jenkins Pipelines](#).

[use_make\(\)](#)

| | |
|---------------|-----------------------------|
| use_lifecycle | <i>Use lifecycle badges</i> |
|---------------|-----------------------------|

Description

This helper:

- Adds lifecycle as a dependency.
- Imports `lifecycle::deprecated()` for use in function arguments.
- Copies the lifecycle badges into man/figures.
- Reminds you how to use the badge syntax.

Learn more at <https://lifecycle.r-lib.org/articles/communicate.html>

Usage

```
use_lifecycle()
```

See Also

[use_lifecycle_badge\(\)](#) to signal the **lifecycle stage** of your package as whole

| | |
|----------|---------------------------|
| use_logo | <i>Use a package logo</i> |
|----------|---------------------------|

Description

This function helps you use a logo in your package:

- Enforces a specific size
- Stores logo image file at man/figures/logo.png
- Produces the markdown text you need in README to include the logo

Usage

```
use_logo(img, geometry = "240x278", retina = TRUE)
```

Arguments

| | |
|----------|--|
| img | The path to an existing image file |
| geometry | a <code>magick::geometry</code> string specifying size. The default assumes that you have a hex logo using spec from http://hexb.in/sticker.html . |
| retina | TRUE, the default, scales the image on the README, assuming that geometry is double the desired size. |

Examples

```
## Not run:  
use_logo("usethis.png")  
  
## End(Not run)
```

| | |
|----------|------------------------|
| use_make | <i>Create Makefile</i> |
|----------|------------------------|

Description

use_make() adds a basic Makefile to the project root directory.

Usage

```
use_make()
```

See Also

The [documentation for GNU Make](#).

| | |
|---------------|------------------------------|
| use_namespace | <i>Use a basic NAMESPACE</i> |
|---------------|------------------------------|

Description

If roxygen is TRUE generates an empty NAMESPACE that exports nothing; you'll need to explicitly export functions with @export. If roxygen is FALSE, generates a default NAMESPACE that exports all functions except those that start with ..

Usage

```
use_namespace(roxygen = TRUE)
```

Arguments

roxygen Do you plan to manage NAMESPACE with roxygen2?

See Also

The [namespace chapter](#) of [R Packages](#).

| | |
|-------------|--------------------------------|
| use_news_md | <i>Create a simple NEWS.md</i> |
|-------------|--------------------------------|

Description

This creates a basic NEWS.md in the root directory.

Usage

```
use_news_md(open = rlang::is_interactive())
```

Arguments

| | |
|------|---|
| open | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |
|------|---|

See Also

The [other markdown files section](#) of [R Packages](#).

| | |
|-------------|----------------------------------|
| use_package | <i>Depend on another package</i> |
|-------------|----------------------------------|

Description

`use_package()` adds a CRAN package dependency to DESCRIPTION and offers a little advice about how to best use it. `use_dev_package()` adds a dependency on an in-development package, adding the dev repo to Remotes so it will be automatically installed from the correct location. There is no helper to remove a dependency: to do that, simply remove that package from your DESCRIPTION file.

`use_package()` exists to support a couple of common maneuvers:

- Add a dependency to Imports or Suggests or LinkingTo.
- Add a minimum version to a dependency.
- Specify the minimum supported version for R.

`use_package()` probably works for slightly more exotic modifications, but at some point, you should edit DESCRIPTION yourself by hand. There is no intention to account for all possible edge cases.

Usage

```
use_package(package, type = "Imports", min_version = NULL)
```

```
use_dev_package(package, type = "Imports", remote = NULL)
```

Arguments

| | |
|-------------|--|
| package | Name of package to depend on. |
| type | Type of dependency: must be one of "Imports", "Depends", "Suggests", "Enhances", or "LinkingTo" (or unique abbreviation). Matching is case insensitive. |
| min_version | Optionally, supply a minimum version for the package. Set to TRUE to use the currently installed version or use a version string suitable for <code>numeric_version()</code> , such as "2.5.0". |
| remote | By default, an OWNER/REPO GitHub remote is inserted. Optionally, you can supply a character string to specify the remote, e.g. "gitlab::jimhester/covr", using any syntax supported by the remotes package . |

See Also

The [dependencies section](#) of [R Packages](#).

Examples

```
## Not run:
use_package("ggplot2")
use_package("dplyr", "suggests")
use_dev_package("glue")

# Depend on R version 4.1
use_package("R", type = "Depends", min_version = "4.1")

## End(Not run)
```

| | |
|-----------------|------------------------------------|
| use_package_doc | <i>Package-level documentation</i> |
|-----------------|------------------------------------|

Description

Adds a dummy .R file that will cause roxygen2 to generate basic package-level documentation. If your package is named "foo", this will make help available to the user via ?foo or package?foo. Once you call `devtools::document()`, roxygen2 will flesh out the .Rd file using data from the DESCRIPTION. That ensures you don't need to repeat (and remember to update!) the same information in multiple places. This .R file is also a good place for roxygen directives that apply to the whole package (vs. a specific function), such as global namespace tags like `@importFrom`.

Usage

```
use_package_doc(open = rlang::is_interactive())
```

Arguments

| | |
|------|---|
| open | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |
|------|---|

See Also

The [documentation chapter](#) of **R Packages**

| | |
|----------|--|
| use_pipe | <i>Use magrittr's pipe in your package</i> |
|----------|--|

Description

Does setup necessary to use magrittr's pipe operator, %>% in your package. This function requires the use of **roxygen2**.

- Adds magrittr to "Imports" in DESCRIPTION.
- Imports the pipe operator specifically, which is necessary for internal use.
- Exports the pipe operator, if `export = TRUE`, which is necessary to make %>% available to the users of your package.

Usage

```
use_pipe(export = TRUE)
```

Arguments

| | |
|--------|---|
| export | If TRUE, the file R/utils-pipe.R is added, which provides the roxygen template to import and re-export %>%. If FALSE, the necessary roxygen directive is added, if possible, or otherwise instructions are given. |
|--------|---|

Examples

```
## Not run:
use_pipe()

## End(Not run)
```

| | |
|-------------|--------------------|
| use_pkgdown | <i>Use pkgdown</i> |
|-------------|--------------------|

Description

pkgdown makes it easy to turn your package into a beautiful website. `usethis` provides two functions to help you use **pkgdown**:

- `use_pkgdown()`: creates a **pkgdown** config file and adds relevant files or directories to `.Rbuildignore` and `.gitignore`.
- `use_pkgdown_github_pages()`: implements the GitHub setup needed to automatically publish your **pkgdown** site to GitHub pages:

- (first, it calls `use_pkgdown()`)
- `use_github_pages()` prepares to publish the pkgdown site from the gh-pages branch
- `use_github_action("pkgdown")` configures a GitHub Action to automatically build the pkgdown site and deploy it via GitHub Pages
- The pkgdown site's URL is added to the pkgdown configuration file, to the URL field of DESCRIPTION, and to the GitHub repo.
- Packages owned by certain GitHub organizations (tidyverse, r-lib, and tidymodels) get some special treatment, in terms of anticipating the (eventual) site URL and the use of a pkgdown template.

Usage

```
use_pkgdown(config_file = "_pkgdown.yml", destdir = "docs")
```

```
use_pkgdown_github_pages()
```

Arguments

| | |
|--------------------------|--|
| <code>config_file</code> | Path to the pkgdown yaml config file, relative to the project. |
| <code>destdir</code> | Target directory for pkgdown docs. |

See Also

<https://pkgdown.r-lib.org/articles/pkgdown.html#configuration>

| | |
|--------------------|---------------------------------------|
| <code>use_r</code> | <i>Create or edit R or test files</i> |
|--------------------|---------------------------------------|

Description

This pair of functions makes it easy to create paired R and test files, using the convention that the tests for `R/foofy.R` should live in `tests/testthat/test-foofy.R`. You can use them to create new files from scratch by supplying name, or if you use RStudio, you can call to create (or navigate to) the companion file based on the currently open file. This also works when a test snapshot file is active, i.e. if you're looking at `tests/testthat/_snaps/foofy.md`, `use_r()` or `use_test()` take you to `R/foofy.R` or `tests/testthat/test-foofy.R`, respectively.

Usage

```
use_r(name = NULL, open = rlang::is_interactive())
```

```
use_test(name = NULL, open = rlang::is_interactive())
```

Arguments

| | |
|-------------------|--|
| <code>name</code> | Either a string giving a file name (without directory) or NULL to take the name from the currently open file in RStudio. |
| <code>open</code> | Whether to open the file for interactive editing. |

Renaming files in an existing package

Here are some tips on aligning file names across R/ and tests/testthat/ in an existing package that did not necessarily follow this convention before.

This script generates a data frame of R/ and test files that can help you identify missed opportunities for pairing:

```
library(fs)
library(tidyverse)

bind_rows(
  tibble(
    type = "R",
    path = dir_ls("R/", regexp = "\\.[Rr]$"),
    name = as.character(path_ext_remove(path_file(path))),
  ),
  tibble(
    type = "test",
    path = dir_ls("tests/testthat/", regexp = "/test[^/]+\\.[Rr]$"),
    name = as.character(path_ext_remove(str_remove(path_file(path), "^test[-_]"))),
  )
) |>
pivot_wider(names_from = type, values_from = path) |>
print(n = Inf)
```

The `rename_files()` function can also be helpful.

See Also

- The [testing](#) and [R code](#) chapters of [R Packages](#).
- [use_test_helper\(\)](#) to create a testthat helper file.

Examples

```
## Not run:
# create a new .R file below R/
use_r("coolstuff")

# if `R/coolstuff.R` is active in a supported IDE, you can now do:
use_test()

# if `tests/testthat/test-coolstuff.R` is active in a supported IDE, you can
# return to `R/coolstuff.R` with:
use_r()

## End(Not run)
```

| | |
|----------|--|
| use_rcpp | <i>Use C, C++, RcppArmadillo, or RcppEigen</i> |
|----------|--|

Description

Adds infrastructure commonly needed when using compiled code:

- Creates `src/`
- Adds required packages to DESCRIPTION
- May create an initial placeholder `.c` or `.cpp` file
- Creates Makevars and Makevars.win files (`use_rcpp_armadillo()` only)

Usage

```
use_rcpp(name = NULL)
```

```
use_rcpp_armadillo(name = NULL)
```

```
use_rcpp_eigen(name = NULL)
```

```
use_c(name = NULL)
```

Arguments

| | |
|------|--|
| name | Either a string giving a file name (without directory) or NULL to take the name from the currently open file in RStudio. |
|------|--|

| | |
|----------------|----------------------------|
| use_readme_rmd | <i>Create README files</i> |
|----------------|----------------------------|

Description

Creates skeleton README files with possible stubs for

- a high-level description of the project/package and its goals
- R code to install from GitHub, if GitHub usage detected
- a basic example

Use `Rmd` if you want a rich intermingling of code and output. Use `md` for a basic README. `README.Rmd` will be automatically added to `.Rbuildignore`. The resulting README is populated with default YAML frontmatter and R fenced code blocks (`md`) or chunks (`Rmd`).

If you use `Rmd`, you'll still need to render it regularly, to keep `README.md` up-to-date. `devtools::build_readme()` is handy for this. You could also use GitHub Actions to re-render `README.Rmd` every time you

push. An example workflow can be found in the examples/ directory here: <https://github.com/r-lib/actions/>.

If the current project is a Git repo, then `use_readme_rmd()` automatically configures a pre-commit hook that helps keep `README.Rmd` and `README.md`, synchronized. The hook creates friction if you try to commit when `README.Rmd` has been edited more recently than `README.md`. If this hook causes more problems than it solves for you, it is implemented in `.git/hooks/pre-commit`, which you can modify or even delete.

Usage

```
use_readme_rmd(open = rlang::is_interactive())
```

```
use_readme_md(open = rlang::is_interactive())
```

Arguments

`open` Open the newly created file for editing? Happens in RStudio, if applicable, or via `utils::file.edit()` otherwise.

See Also

The [other markdown files section](#) of [R Packages](#).

Examples

```
## Not run:
use_readme_rmd()
use_readme_md()

## End(Not run)
```

| | |
|--------------------------------|---|
| <code>use_release_issue</code> | <i>Create a release checklist in a GitHub issue</i> |
|--------------------------------|---|

Description

When preparing to release a package to CRAN there are quite a few steps that need to be performed, and some of the steps can take multiple hours. This function creates a checklist in a GitHub issue to:

- Help you keep track of where you are in the process
- Feel a sense of satisfaction as you progress towards final submission
- Help watchers of your package stay informed.

The checklist contains a generic set of steps that we've found to be helpful, based on the type of release ("patch", "minor", or "major"). You're encouraged to edit the issue to customize this list to meet your needs.

Customization:

- If you want to consistently add extra bullets for every release, you can include your own custom bullets by providing an (unexported) `release_bullets()` function that returns a character vector. (For historical reasons, `release_questions()` is also supported).
- If you want to check additional packages in the revdep check process, provide an (unexported) `release_extra_revdeps()` function that returns a character vector. This is currently only supported for Posit internal check tooling.

Usage

```
use_release_issue(version = NULL)
```

Arguments

`version` Optional version number for release. If unspecified, you can make an interactive choice.

Examples

```
## Not run:  
use_release_issue("2.0.0")  
  
## End(Not run)
```

use_revdep *Reverse dependency checks*

Description

Performs set up for checking the reverse dependencies of an R package, as implemented by the `revdepcheck` package:

- Creates `revdep/` directory and adds it to `.Rbuildignore`
- Populates `revdep/.gitignore` to prevent tracking of various revdep artefacts
- Prompts user to run the checks with `revdepcheck::revdep_check()`

Usage

```
use_revdep()
```

`use_rmarkdown_template`*Add an RMarkdown Template*

Description

Adds files and directories necessary to add a custom rmarkdown template to RStudio. It creates:

- `inst/rmarkdown/templates/{{template_dir}}`. Main directory.
- `skeleton/skeleton.Rmd`. Your template Rmd file.
- `template.yml` with basic information filled in.

Usage

```
use_rmarkdown_template(  
  template_name = "Template Name",  
  template_dir = NULL,  
  template_description = "A description of the template",  
  template_create_dir = FALSE  
)
```

Arguments

| | |
|-----------------------------------|---|
| <code>template_name</code> | The name as printed in the template menu. |
| <code>template_dir</code> | Name of the directory the template will live in within <code>inst/rmarkdown/templates</code> . If none is provided by the user, it will be created from <code>template_name</code> . |
| <code>template_description</code> | Sets the value of <code>description</code> in <code>template.yml</code> . |
| <code>template_create_dir</code> | Sets the value of <code>create_dir</code> in <code>template.yml</code> . |

Examples

```
## Not run:  
use_rmarkdown_template()  
  
## End(Not run)
```

| | |
|----------------|-----------------------------------|
| use_roxygen_md | <i>Use roxygen2 with markdown</i> |
|----------------|-----------------------------------|

Description

If you are already using roxygen2, but not with markdown, you'll need to use `roxygen2md` to convert existing Rd expressions to markdown. The conversion is not perfect, so make sure to check the results.

Usage

```
use_roxygen_md(overwrite = FALSE)
```

Arguments

| | |
|-----------|---|
| overwrite | Whether to overwrite an existing Roxygen field in DESCRIPTION with "list(markdown = TRUE)". |
|-----------|---|

| | |
|-------------|---|
| use_rstudio | <i>Add RStudio Project infrastructure</i> |
|-------------|---|

Description

It is likely that you want to use `create_project()` or `create_package()` instead of `use_rstudio()`! Both `create_*`() functions can add RStudio Project infrastructure to a pre-existing project or package. `use_rstudio()` is mostly for internal use or for those creating a usethis-like package for their organization. It does the following in the current project, often after executing `proj_set(..., force = TRUE)`:

- Creates an `.Rproj` file
- Adds RStudio files to `.gitignore`
- Adds RStudio files to `.Rbuildignore`, if project is a package

Usage

```
use_rstudio(line_ending = c("posix", "windows"), reformat = TRUE)
```

Arguments

| | |
|-------------|---|
| line_ending | Line ending |
| reformat | If TRUE, the <code>.Rproj</code> is setup with common options that reformat files on save: adding a trailing newline, trimming trailing whitespace, and setting the line-ending. This is best practice for new projects. If FALSE, these options are left unset, which is more appropriate when you're contributing to someone else's project that does not have its own <code>.Rproj</code> file. |

 use_rstudio_preferences

Set global RStudio preferences

Description

This function allows you to set global RStudio preferences, achieving the same effect programmatically as clicking buttons in RStudio's Global Options. You can find a list of configurable properties at https://docs.posit.co/ide/server-pro/reference/session_user_settings.html.

Usage

```
use_rstudio_preferences(...)
```

Arguments

... `<dynamic-dots>` Property-value pairs.

Value

A named list of the previous values, invisibly.

 use_spell_check

Use spell check

Description

Adds a unit test to automatically run a spell check on documentation and, optionally, vignettes during R CMD check, using the `spelling` package. Also adds a WORDLIST file to the package, which is a dictionary of whitelisted words. See `spelling::wordlist` for details.

Usage

```
use_spell_check(vignettes = TRUE, lang = "en-US", error = FALSE)
```

Arguments

| | |
|-----------|---|
| vignettes | Logical, TRUE to spell check all rmd and rnw files in the vignettes/ folder. |
| lang | Preferred spelling language. Usually either "en-US" or "en-GB". |
| error | Logical, indicating whether the unit test should fail if spelling errors are found. Defaults to FALSE, which does not error, but prints potential spelling errors |

| | |
|----------------|--|
| use_standalone | <i>Use a standalone file from another repo</i> |
|----------------|--|

Description

A "standalone" file implements a minimum set of functionality in such a way that it can be copied into another package. `use_standalone()` makes it easy to get such a file into your own repo.

It always overwrites an existing standalone file of the same name, making it easy to update previously imported code.

Usage

```
use_standalone(repo_spec, file = NULL, ref = NULL, host = NULL)
```

Arguments

| | |
|-----------|--|
| repo_spec | A string identifying the GitHub repo in one of these forms: <ul style="list-style-type: none"> • Plain OWNER/REPO spec • Browser URL, such as "https://github.com/OWNER/REPO" • HTTPS Git URL, such as "https://github.com/OWNER/REPO.git" • SSH Git URL, such as "git@github.com:OWNER/REPO.git" |
| file | Name of standalone file. The standalone- prefix and file extension are optional. If omitted, will allow you to choose from the standalone files offered by that repo. |
| ref | The name of a branch, tag, or commit. By default, the file at path will be copied from its current state in the repo's default branch. This is extracted from repo_spec when user provides a URL. |
| host | GitHub host to target, passed to the <code>.api_url</code> argument of <code>gh::gh()</code> . If repo_spec is a URL, host is extracted from that. If unspecified, gh defaults to "https://api.github.com", although gh's default can be customised by setting the GITHUB_API_URL environment variable. For a hypothetical GitHub Enterprise instance, either "https://github.acme.com/api/v3" or "https://github.acme.com" is acceptable. |

Supported fields

A standalone file has YAML frontmatter that provides additional information, such as where the file originates from and when it was last updated. Here is an example:

```
---
repo: r-lib/rlang
file: standalone-types-check.R
last-updated: 2023-03-07
license: https://unlicense.org
dependencies: standalone-obj-type.R
```

```
imports: rlang (>= 1.1.0)
---
```

Two of these fields are consulted by `use_standalone()`:

- `dependencies`: A file or a list of files in the same repo that the standalone file depends on. These files are retrieved automatically by `use_standalone()`.
- `imports`: A package or list of packages that the standalone file depends on. A minimal version may be specified in parentheses, e.g. `rlang (>= 1.0.0)`. These dependencies are passed to `use_package()` to ensure they are included in the `Imports:` field of the DESCRIPTION file.

Note that lists are specified with standard YAML syntax, using square brackets, for example: `imports: [rlang (>= 1.0.0), purrr]`.

Examples

```
## Not run:
use_standalone("r-lib/rlang", file = "types-check")
use_standalone("r-lib/rlang", file = "types-check", ref = "standalone-dep")

## End(Not run)
```

use_template

Use a usethis-style template

Description

Creates a file from data and a template found in a package. Provides control over file name, the addition to `.Rbuildignore`, and opening the file for inspection.

Usage

```
use_template(
  template,
  save_as = template,
  data = list(),
  ignore = FALSE,
  open = FALSE,
  package = "usethis"
)
```

Arguments

| | |
|-----------------------|--|
| <code>template</code> | Path to template file relative to <code>templates/</code> directory within package; see details. |
| <code>save_as</code> | Path of file to create, relative to root of active project. Defaults to <code>template</code> |
| <code>data</code> | A list of data passed to the template. |

| | |
|---------|---|
| ignore | Should the newly created file be added to .Rbuildignore? |
| open | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |
| package | Name of the package where the template is found. |

Details

This function can be used as the engine for a templating function in other packages. The `template` argument is used along with the `package` argument to derive the path to your template file; it will be expected at `fs::path_package(package = package, "templates", template)`. We use `fs::path_package()` instead of `base::system.file()` so that path construction works even in a development workflow, e.g., works with `devtools::load_all()` or `pkgload::load_all()`. *Note this describes the behaviour of `fs::path_package()` in `fs v1.2.7.9001` and higher.*

To interpolate your data into the template, supply a list using the `data` argument. Internally, this function uses `whisker::whisker.render()` to combine your template file with your data.

Value

A logical vector indicating if file was modified.

Examples

```
## Not run:
# Note: running this will write `NEWS.md` to your working directory
use_template(
  template = "NEWS.md",
  data = list(Package = "acme", Version = "1.2.3"),
  package = "usethis"
)

## End(Not run)
```

| | |
|--------------|---|
| use_testthat | <i>Sets up overall testing infrastructure</i> |
|--------------|---|

Description

Creates `tests/testthat/`, `tests/testthat.R`, and adds the `testthat` package to the `Suggests` field. Learn more in <https://r-pkgs.org/testing-basics.html>

Usage

```
use_testthat(edition = NULL, parallel = FALSE)
```

Arguments

| | |
|----------|--|
| edition | testthat edition to use. Defaults to the latest edition, i.e. the major version number of the currently installed testthat. |
| parallel | Should tests be run in parallel? This feature appeared in testthat 3.0.0; see https://testthat.r-lib.org/articles/parallel.html for details and caveats. |

See Also

[use_test\(\)](#) to create individual test files

Examples

```
## Not run:
use_testthat()

use_test()

use_test("something-management")

## End(Not run)
```

| | |
|-----------------|--|
| use_test_helper | <i>Create or edit a test helper file</i> |
|-----------------|--|

Description

This function creates (or opens) a test helper file, typically `tests/testthat/helper.R`. Test helper files are executed at the beginning of every automated test run and are also executed by `load_all()`. A helper file is a great place to define test helper functions for use throughout your test suite, such as a custom expectation.

Usage

```
use_test_helper(name = NULL, open = rlang::is_interactive())
```

Arguments

| | |
|------|---|
| name | Can be used to specify the optional "SLUG" in <code>tests/testthat/helper-SLUG.R</code> . |
| open | Whether to open the file for interactive editing. |

See Also

- [use_test\(\)](#) to create a test file.
- The testthat vignette on special files `vignette("special-files", package = "testthat")`.

Examples

```
## Not run:  
use_test_helper()  
use_test_helper("mocks")  
  
## End(Not run)
```

| | |
|------------|-----------------------------------|
| use_tibble | <i>Prepare to return a tibble</i> |
|------------|-----------------------------------|

Description

[Questioning]

Does minimum setup such that a tibble returned by your package is handled using the tibble method for generics like `print()` or `[-`. Presumably you care about this if you've chosen to store and expose an object with class `tbl_df`. Specifically:

- Check that the active package uses `roxygen2`
- Add the tibble package to "Imports" in DESCRIPTION
- Prepare the `roxygen` directive necessary to import at least one function from tibble:
 - If possible, the directive is inserted into existing package-level documentation, i.e. the `roxygen` snippet created by `use_package_doc()`
 - Otherwise, we issue advice on where the user should add the directive

This is necessary when your package returns a stored data object that has class `tbl_df`, but the package code does not make direct use of functions from the tibble package. If you do nothing, the tibble namespace is not necessarily loaded and your tibble may therefore be printed and subsetted like a base `data.frame`.

Usage

```
use_tibble()
```

Examples

```
## Not run:  
use_tibble()  
  
## End(Not run)
```

`use_tidy_github_actions`*Helpers for tidyverse development*

Description

These helpers follow tidyverse conventions which are generally a little stricter than the defaults, reflecting the need for greater rigor in commonly used packages.

Usage

```
use_tidy_github_actions(ref = NULL)

create_tidy_package(path, copyright_holder = NULL)

use_tidy_description()

use_tidy_dependencies()

use_tidy_contributing()

use_tidy_support()

use_tidy_issue_template()

use_tidy_coc()

use_tidy_github()

use_tidy_logo(geometry = "240x278", retina = TRUE)

use_tidy_upkeep_issue(last_upkeep = last_upkeep_year())
```

Arguments

| | |
|-------------------------------|---|
| <code>ref</code> | Desired Git reference, usually the name of a tag ("v2") or branch ("main"). Other possibilities include a commit SHA ("d1c516d") or "HEAD" (meaning "tip of remote's default branch"). If not specified, defaults to the latest published release of r-lib/actions (https://github.com/r-lib/actions/releases). |
| <code>path</code> | A path. If it exists, it is used. If it does not exist, it is created, provided that the parent path exists. |
| <code>copyright_holder</code> | Name of the copyright holder or holders. This defaults to "{package name} authors"; you should only change this if you use a CLA to assign copyright to a single entity. |

| | |
|-------------|--|
| geometry | a <code>magick::geometry</code> string specifying size. The default assumes that you have a hex logo using spec from http://hexb.in/sticker.html . |
| retina | TRUE, the default, scales the image on the README, assuming that geometry is double the desired size. |
| last_upkeep | Year of last upkeep. By default, the <code>Config/usethis/last-upkeep</code> field in DESCRIPTION is consulted for this, if it's defined. If there's no information on the last upkeep, the issue will contain the full checklist. |

Details

- `use_tidy_github_actions()`: Sets up the following workflows using **GitHub Actions**:
 - Run R CMD check on the current release, devel, and four previous versions of R. The build matrix also ensures R CMD check is run at least once on each of the three major operating systems (Linux, macOS, and Windows).
 - Report test coverage.
 - Build and deploy a pkgdown site.
 - Check the formatting of incoming pull requests with Air and suggest fixes as necessary. This is how the tidyverse team checks its packages, but it is overkill for less widely used packages. For R CMD check, consider using the more streamlined workflow set up by `use_github_action("check-standard")`.
- `create_tidy_package()`: creates a new package, immediately applies as many of the tidyverse conventions as possible, issues a few reminders, and activates the new package.
- `use_tidy_dependencies()`: sets up standard dependencies used by all tidyverse packages (except packages that are designed to be dependency free).
- `use_tidy_description()`: puts fields in standard order and alphabetises dependencies.
- `use_tidy_eval()`: imports a standard set of helpers to facilitate programming with the tidy eval toolkit.
- `use_tidy_style()`: styles source code according to the **tidyverse style guide**. This function will overwrite files! See below for usage advice.
- `use_tidy_contributing()`: adds standard tidyverse contributing guidelines.
- `use_tidy_issue_template()`: adds a standard tidyverse issue template.
- `use_tidy_release_test_env()`: updates the test environment section in `cran-comments.md`.
- `use_tidy_support()`: adds a standard description of support resources for the tidyverse.
- `use_tidy_coc()`: equivalent to `use_code_of_conduct()`, but puts the document in a `.github/` subdirectory.
- `use_tidy_github()`: convenience wrapper that calls `use_tidy_contributing()`, `use_tidy_issue_template()`, `use_tidy_support()`, `use_tidy_coc()`.
- `use_tidy_github_labels()` calls `use_github_labels()` to implement tidyverse conventions around GitHub issue label names and colours.
- `use_tidy_upkeep_issue()` creates an issue containing a checklist of actions to bring your package up to current tidyverse standards. Also records the current date in the `Config/usethis/last-upkeep` field in DESCRIPTION.
- `use_tidy_logo()` calls `use_logo()` on the appropriate hex sticker PNG file at <https://github.com/rstudio/hex-stickers>.

use_tidy_thanks *Identify contributors via GitHub activity*

Description

Derives a list of GitHub usernames, based on who has opened issues or pull requests. Used to populate the acknowledgment section of package release blog posts at <https://www.tidyverse.org/blog/>. If no arguments are given, we retrieve all contributors to the active project since its last (GitHub) release. Unexported helper functions, `releases()` and `ref_df()` can be useful interactively to get a quick look at release tag names and a data frame about refs (defaulting to releases), respectively.

Usage

```
use_tidy_thanks(repo_spec = NULL, from = NULL, to = NULL)
```

Arguments

| | |
|------------------------|--|
| <code>repo_spec</code> | Optional GitHub repo specification in any form accepted for the <code>repo_spec</code> argument of <code>create_from_github()</code> (plain spec or a browser or Git URL). A URL specification is the only way to target a GitHub host other than "github.com", which is the default. |
| <code>from, to</code> | GitHub ref (i.e., a SHA, tag, or release) or a timestamp in ISO 8601 format, specifying the start or end of the interval of interest, in the sense of <code>[from, to]</code> . Examples: "08a560d", "v1.3.0", "2018-02-24T00:13:45Z", "2018-05-01". When <code>from = NULL</code> , <code>to = NULL</code> , we set <code>from</code> to the timestamp of the most recent (GitHub) release. Otherwise, NULL means "no bound". |

Value

A character vector of GitHub usernames, invisibly.

Examples

```
## Not run:
# active project, interval = since the last release
use_tidy_thanks()

# active project, interval = since a specific datetime
use_tidy_thanks(from = "2020-07-24T00:13:45Z")

# r-lib/usethis, interval = since a certain date
use_tidy_thanks("r-lib/usethis", from = "2020-08-01")

# r-lib/usethis, up to a specific release
use_tidy_thanks("r-lib/usethis", from = NULL, to = "v1.1.0")

# r-lib/usethis, since a specific commit, up to a specific date
```

```
use_tidy_thanks("r-lib/usethis", from = "08a560d", to = "2018-05-14")

# r-lib/usethis, but with copy/paste of a browser URL
use_tidy_thanks("https://github.com/r-lib/usethis")

## End(Not run)
```

use_tutorial

Create a learnr tutorial

Description

Creates a new tutorial below `inst/tutorials/`. Tutorials are interactive R Markdown documents built with the [learnr package](#). `use_tutorial()` does this setup:

- Adds learnr to Suggests in DESCRIPTION.
- Gitignores `inst/tutorials/*.html` so you don't accidentally track rendered tutorials.
- Creates a new `.Rmd` tutorial from a template and, optionally, opens it for editing.
- Adds new `.Rmd` to `.Rbuildignore`.

Usage

```
use_tutorial(name, title, open = rlang::is_interactive())
```

Arguments

| | |
|-------|---|
| name | Base for file name to use for new <code>.Rmd</code> tutorial. Should consist only of numbers, letters, <code>_</code> and <code>-</code> . We recommend using lower case. |
| title | The human-facing title of the tutorial. |
| open | Open the newly created file for editing? Happens in RStudio, if applicable, or via <code>utils::file.edit()</code> otherwise. |

See Also

The [learnr package documentation](#).

Examples

```
## Not run:
use_tutorial("learn-to-do-stuff", "Learn to do stuff")

## End(Not run)
```

use_upkeep_issue *Create an upkeep checklist in a GitHub issue*

Description

This opens an issue in your package repository with a checklist of tasks for regular maintenance of your package. This is a fairly opinionated list of tasks but we believe taking care of them will generally make your package better, easier to maintain, and more enjoyable for your users. Some of the tasks are meant to be performed only once (and once completed shouldn't show up in subsequent lists), and some should be reviewed periodically. The tidyverse team uses a similar function `use_tidy_upkeep_issue()` for our annual package Spring Cleaning.

Usage

```
use_upkeep_issue(year = NULL)
```

Arguments

`year` Year you are performing the upkeep, used in the issue title. Defaults to current year

Examples

```
## Not run:
use_upkeep_issue()

## End(Not run)
```

use_version *Increment package version*

Description

usethis supports semantic versioning, which is described in more detail in the [version section](#) of [R Packages](#). A version number breaks down like so:

```
<major>.<minor>.<patch>              (released version)
<major>.<minor>.<patch>.<dev>        (dev version)
```

`use_version()` increments the "Version" field in DESCRIPTION, adds a new heading to NEWS.md (if it exists), commits those changes (if package uses Git), and optionally pushes (if safe to do so). It makes the same update to a line like `PKG_version = "x.y.z"`; in `src/version.c` (if it exists).

`use_dev_version()` increments to a development version, e.g. from 1.0.0 to 1.0.0.9000. If the existing version is already a development version with four components, it does nothing. Thin wrapper around `use_version()`.

Usage

```
use_version(which = NULL, push = FALSE)
```

```
use_dev_version(push = FALSE)
```

Arguments

which A string specifying which level to increment, one of: "major", "minor", "patch", "dev". If NULL, user can choose interactively.

push If TRUE, also attempts to push the commits to the remote branch.

See Also

The [version section](#) of [R Packages](#).

Examples

```
## Not run:  
## for interactive selection, do this:  
use_version()  
  
## request a specific type of increment  
use_version("minor")  
use_dev_version()  
  
## End(Not run)
```

use_vignette

Create a vignette or article

Description

Creates a new vignette or article in vignettes/. Articles are a special type of vignette that appear on pkgdown websites, but are not included in the package itself (because they are added to .Rbuildignore automatically).

Usage

```
use_vignette(name, title = NULL)
```

```
use_article(name, title = NULL)
```

Arguments

| | |
|-------|---|
| name | File name to use for new vignette. Should consist only of numbers, letters, _ and -. Lower case is recommended. Can include the ".Rmd" or ".qmd" file extension, which also dictates whether to place an R Markdown or Quarto vignette. R Markdown (".Rmd") is the current default, but it is anticipated that Quarto (".qmd") will become the default in the future. |
| title | The title of the vignette. If not provided, a title is generated from name. |

General setup

- Adds needed packages to DESCRIPTION.
- Adds inst/doc to .gitignore so built vignettes aren't tracked.
- Adds vignettes/*.html and vignettes/*.R to .gitignore so you never accidentally track rendered vignettes.
- For *.qmd, adds Quarto-related patterns to .gitignore and .Rbuildignore.

See Also

- The [vignettes chapter](#) of [R Packages](#)
- The pkgdown vignette on Quarto: `vignette("quarto", package = "pkgdown")`
- The quarto (as in the R package) vignette on HTML vignettes: `vignette("hello", package = "quarto")`

Examples

```
## Not run:
use_vignette("how-to-do-stuff", "How to do stuff")
use_vignette("r-markdown-is-classic.Rmd", "R Markdown is classic")
use_vignette("quarto-is-cool.qmd", "Quarto is cool")

## End(Not run)
```

zip-utils

Download and unpack a ZIP file

Description

Functions to download and unpack a ZIP file into a local folder of files, with very intentional default behaviour. Useful in pedagogical settings or anytime you need a large audience to download a set of files quickly and actually be able to find them. After download, the new folder is opened in a new session of the user's IDE, if possible, or in the default file manager provided by the operating system. The underlying helpers are documented in [use_course_details](#).

Usage

```

use_course(url, destdir = getOption("usethis.destdir"))

use_zip(
  url,
  destdir = getwd(),
  cleanup = if (rlang::is_interactive()) NA else FALSE
)

```

Arguments

| | |
|----------------------|--|
| <code>url</code> | Link to a ZIP file containing the materials. To reduce the chance of typos in live settings, these shorter forms are accepted: <ul style="list-style-type: none"> • GitHub repo spec: "OWNER/REPO". Equivalent to <code>https://github.com/OWNER/REPO/DEFAULT_BRANCH</code>. • bit.ly, pos.it, or rstd.io shortlinks: "bit.ly/xxx-yyy-zzz", "pos.it/foofy" or "rstd.io/foofy". The instructor must then arrange for the shortlink to point to a valid download URL for the target ZIP file. The helper <code>create_download_url()</code> helps to create such URLs for GitHub, DropBox, and Google Drive. |
| <code>destdir</code> | Destination for the new folder. Defaults to the location stored in the global option <code>usethis.destdir</code> , if defined, or to the user's Desktop or similarly conspicuous place otherwise. |
| <code>cleanup</code> | Whether to delete the original ZIP file after unpacking its contents. In an interactive setting, NA leads to a menu where user can approve the deletion (or decline). |

Value

Path to the new directory holding the unpacked ZIP file, invisibly.

Functions

- `use_course()`: Designed with live workshops in mind. Includes intentional friction to highlight the download destination. Workflow:
 - User executes, e.g., `use_course("bit.ly/xxx-yyy-zzz")`.
 - User is asked to notice and confirm the location of the new folder. Specify `destdir` or configure the `"usethis.destdir"` option to prevent this.
 - User is asked if they'd like to delete the ZIP file.
 - If possible, the new folder is launched in a new session of the user's IDE. Otherwise, the folder is opened in the file manager, e.g. Finder on macOS or File Explorer on Windows.
- `use_zip()`: More useful in day-to-day work. Downloads in current working directory, by default, and allows cleanup behaviour to be specified.

Examples

```

## Not run:
# download the source of usethis from GitHub, behind a bit.ly shortlink
use_course("bit.ly/usethis-shortlink-example")

```

```
use_course("http://bit.ly/usethis-shortlink-example")

# download the source of rematch2 package from CRAN
use_course("https://cran.r-project.org/bin/windows/contrib/4.5/rematch2_2.1.2.zip")

# download the source of rematch2 package from GitHub, 4 ways
use_course("r-lib/rematch2")
use_course("https://api.github.com/repos/r-lib/rematch2/zipball/HEAD")
use_course("https://api.github.com/repos/r-lib/rematch2/zipball/main")
use_course("https://github.com/r-lib/rematch2/archive/main.zip")

## End(Not run)
```

Index

* **git helpers**

- use_git, [40](#)
- use_git_config, [51](#)
- use_git_hook, [52](#)
- use_git_ignore, [53](#)

* **project functions**

- proj_sitrep, [21](#)
- proj_utils, [22](#)

activates, [8](#), [10](#)

active project, [20](#)

badges, [4](#)

browse-this, [5](#)

browse_circleci (browse-this), [5](#)

browse_cran (browse-this), [5](#)

browse_github (browse-this), [5](#)

browse_github_actions (browse-this), [5](#)

browse_github_issues (browse-this), [5](#)

browse_github_pulls (browse-this), [5](#)

browse_github_pulls(), [27](#)

browse_package (browse-this), [5](#)

browse_project (browse-this), [5](#)

covr::codecov(), [43](#)

create_download_url(), [81](#)

create_from_github, [7](#)

create_from_github(), [30](#), [76](#)

create_github_token (github-token), [14](#)

create_package, [9](#)

create_package(), [32](#), [39](#), [67](#)

create_project (create_package), [9](#)

create_project(), [67](#)

create_quarto_project (create_package),
[9](#)

create_tidy_package

(use_tidy_github_actions), [74](#)

create_tidy_package(), [10](#)

data(), [37](#)

devtools::submit_cran(), [50](#)

edit, [11](#)

edit_git_config (edit), [11](#)

edit_git_ignore (edit), [11](#)

edit_pkgdown_config (edit), [11](#)

edit_r_buildignore (edit), [11](#)

edit_r_environ (edit), [11](#)

edit_r_makevars (edit), [11](#)

edit_r_profile (edit), [11](#)

edit_r_profile(), [16](#), [29](#), [39](#)

edit_rstudio_prefs (edit), [11](#)

edit_rstudio_snippets (edit), [11](#)

fs::path(), [23](#)

fs::path_home(), [12](#)

functions for working with pull
requests, [7](#)

gert::git_config(), [51](#)

gert::git_config_global_set(), [51](#)

gert::git_config_set(), [51](#)

gh::gh(), [8](#), [15](#), [42](#), [46](#), [69](#)

gh::gh_whoami(), [15](#)

gh_token_help (github-token), [14](#)

gh_token_help(), [8](#), [25](#), [42](#)

git-default-branch, [12](#)

git_default_branch

(git-default-branch), [12](#)

git_default_branch(), [24](#)

git_default_branch_configure

(git-default-branch), [12](#)

git_default_branch_rediscover

(git-default-branch), [12](#)

git_default_branch_rename

(git-default-branch), [12](#)

git_protocol, [16](#)

git_protocol(), [9](#)

git_remotes (use_git_remote), [53](#)

git_sitrep, [17](#)

- git_vaccinate, 18
- github-token, 14
- issue-this, 18
- issue_close_community (issue-this), 18
- issue_reprex_needed (issue-this), 18
- licenses, 19
- lifecycle::deprecated(), 56
- load_all(), 72
- local_project (proj_utils), 22
- magick::geometry, 56, 75
- numeric_version(), 59
- pkgload::load_all(), 55
- pr_fetch (pull-requests), 24
- pr_finish (pull-requests), 24
- pr_forget (pull-requests), 24
- pr_init (pull-requests), 24
- pr_init(), 7
- pr_merge_main (pull-requests), 24
- pr_pause (pull-requests), 24
- pr_pull (pull-requests), 24
- pr_push (pull-requests), 24
- pr_resume (pull-requests), 24
- pr_view (pull-requests), 24
- proj_activate, 20
- proj_get (proj_utils), 22
- proj_path (proj_utils), 22
- proj_set (proj_utils), 22
- proj_sitrep, 21, 23
- proj_utils, 21, 22
- pull-requests, 24
- quarto::quarto_create_project(), 9
- rename_files, 27
- rename_files(), 62
- reverse dependency checks, 36
- rprofile-helper, 28
- RStudio's preference file, 11
- save(), 37
- spelling, 68
- spelling::wordlist, 68
- tidy_label_colours (use_github_labels), 46
- tidy_label_descriptions (use_github_labels), 46
- tidy_labels (use_github_labels), 46
- tidy_labels_rename (use_github_labels), 46
- tidyverse (use_tidy_github_actions), 74
- ui_silence, 28
- use_addin, 30
- use_agpl3_license (licenses), 19
- use_agpl_license (licenses), 19
- use_air, 31
- use_air(), 43
- use_apache_license (licenses), 19
- use_apl2_license (licenses), 19
- use_article (use_vignette), 79
- use_author, 32
- use_badge (badges), 4
- use_binder_badge (badges), 4
- use_bioc_badge (badges), 4
- use_blank_slate, 33
- use_build_ignore, 34
- use_c (use_rcpp), 63
- use_cc0_license (licenses), 19
- use_ccby_license (licenses), 19
- use_circleci (use_gitlab_ci), 50
- use_circleci_badge (use_gitlab_ci), 50
- use_citation, 34
- use_code_of_conduct, 35
- use_conflicted (rprofile-helper), 28
- use_course (zip-utils), 80
- use_course(), 9, 30
- use_course_details, 80
- use_coverage, 35
- use_covr_ignore (use_coverage), 35
- use_cpp11, 36
- use_cran_badge (badges), 4
- use_cran_comments, 36
- use_data, 37
- use_data(), 40
- use_data_raw (use_data), 37
- use_data_table, 38
- use_description, 38
- use_description(), 10, 32
- use_description_defaults (use_description), 38
- use_dev_package (use_package), 58
- use_dev_version (use_version), 78
- use_devtools (rprofile-helper), 28

[use_directory](#), 40
[use_directory\(\)](#), 35
[use_git](#), 40, 52, 53
[use_git_config](#), 41, 51, 52, 53
[use_git_hook](#), 41, 52, 52, 53
[use_git_ignore](#), 41, 52, 53
[use_git_protocol](#) (git_protocol), 16
[use_git_protocol\(\)](#), 30
[use_git_remote](#), 53
[use_github](#), 41
[use_github\(\)](#), 9
[use_github_action](#), 43
[use_github_action\(\)](#), 5
[use_github_action\(pkgdown\)](#), 49, 61
[use_github_file](#), 45
[use_github_file\(\)](#), 44
[use_github_labels](#), 46
[use_github_links](#), 48
[use_github_links\(\)](#), 41
[use_github_pages](#), 49
[use_github_pages\(\)](#), 61
[use_github_release](#), 50
[use_gitlab_ci](#), 50
[use_gpl3_license](#) (licenses), 19
[use_gpl_license](#) (licenses), 19
[use_import_from](#), 54
[use_jenkins](#), 55
[use_lgpl_license](#) (licenses), 19
[use_lifecycle](#), 56
[use_lifecycle_badge](#) (badges), 4
[use_lifecycle_badge\(\)](#), 56
[use_logo](#), 56
[use_make](#), 57
[use_make\(\)](#), 55
[use_mit_license](#) (licenses), 19
[use_namespace](#), 57
[use_news_md](#), 58
[use_package](#), 58
[use_package\(\)](#), 70
[use_package_doc](#), 59
[use_package_doc\(\)](#), 54, 73
[use_partial_warnings](#) (rprofile-helper), 28
[use_pipe](#), 60
[use_pkgdown](#), 60
[use_pkgdown_github_pages](#) (use_pkgdown), 60
[use_pkgdown_github_pages\(\)](#), 43, 50
[use_posit_cloud_badge](#) (badges), 4
[use_posit_cloud_badge\(\)](#), 5
[use_proprietary_license](#) (licenses), 19
[use_r](#), 61
[use_r_universe_badge](#) (badges), 4
[use_rcpp](#), 63
[use_rcpp_armadillo](#) (use_rcpp), 63
[use_rcpp_eigen](#) (use_rcpp), 63
[use_readme_md](#) (use_readme_rmd), 63
[use_readme_md\(\)](#), 4
[use_readme_rmd](#), 63
[use_readme_rmd\(\)](#), 4
[use_release_issue](#), 64
[use_reprex](#) (rprofile-helper), 28
[use_revdep](#), 65
[use_rmarkdown_template](#), 66
[use_roxygen_md](#), 67
[use_rstudio](#), 67
[use_rstudio\(\)](#), 10
[use_rstudio_preferences](#), 68
[use_spell_check](#), 68
[use_standalone](#), 69
[use_template](#), 70
[use_test](#) (use_r), 61
[use_test\(\)](#), 72
[use_test_helper](#), 72
[use_test_helper\(\)](#), 62
[use_testthat](#), 71
[use_tibble](#), 73
[use_tidy_coc](#) (use_tidy_github_actions), 74
[use_tidy_contributing](#) (use_tidy_github_actions), 74
[use_tidy_dependencies](#) (use_tidy_github_actions), 74
[use_tidy_description](#) (use_tidy_github_actions), 74
[use_tidy_github](#) (use_tidy_github_actions), 74
[use_tidy_github_actions](#), 74
[use_tidy_github_labels](#) (use_github_labels), 46
[use_tidy_github_labels\(\)](#), 75
[use_tidy_issue_template](#) (use_tidy_github_actions), 74
[use_tidy_logo](#) (use_tidy_github_actions), 74
[use_tidy_support](#)

- [\(use_tidy_github_actions\)](#), [74](#)
- [use_tidy_thanks](#), [76](#)
- [use_tidy_upkeep_issue](#)
 - [\(use_tidy_github_actions\)](#), [74](#)
- [use_tidy_upkeep_issue\(\)](#), [78](#)
- [use_tutorial](#), [77](#)
- [use_upkeep_issue](#), [78](#)
- [use_usethis \(rprofile-helper\)](#), [28](#)
- [use_version](#), [78](#)
- [use_vignette](#), [79](#)
- [use_vignette\(\)](#), [40](#)
- [use_zip \(zip-utils\)](#), [80](#)
- [usethis_options](#), [29](#)
- [utils::file.edit\(\)](#), [30](#), [36](#), [37](#), [44](#), [45](#), [58](#),
[59](#), [64](#), [71](#), [77](#)
- [utils::person](#), [32](#)

- [whisker::whisker.render\(\)](#), [71](#)
- [with_project \(proj_utils\)](#), [22](#)

- [zip-utils](#), [80](#)