Package ‘xfun’

January 18, 2026

Type Package

Title Supporting Functions for Packages Maintained by 'Yihui Xie'

Version 0.56

Description Miscellaneous functions commonly used in other packages maintained by 'Yihui Xie'.

Depends R (>= 3.2.0)

Imports grDevices, stats, tools

Suggests testit, parallel, codetools, methods, rstudioapi, tinytex (>= 0.30), mime, litedown (>= 0.6), commonmark, knitr (>= 1.50), remotes, pak, curl, xml2, jsonlite, magick, yaml, data.table, qs2

License MIT + file LICENSE

URL `https://github.com/yihui/xfun`

BugReports `https://github.com/yihui/xfun/issues`

Encoding UTF-8

RoxygenNote 7.3.3

VignetteBuilder litedown

NeedsCompilation yes

Author Yihui Xie [aut, cre, cph] (ORCID: <`https://orcid.org/0000-0003-0645-5666`>, URL: https://yihui.org),
Wush Wu [ctb],
Daijiang Li [ctb],
Xianying Tan [ctb],
Salim Brüggemann [ctb] (ORCID: <`https://orcid.org/0000-0002-5329-5987`>),
Christophe Dervieux [ctb]

Maintainer Yihui Xie <`xie@yihui.name`>

Repository CRAN

Date/Publication 2026-01-18 06:10:03 UTC

Contents

`alnum_id` *Generate ID strings*

Description

Substitute certain (by default, non-alphanumeric) characters with dashes and remove extra dashes at both ends to generate ID strings. This function is intended for generating IDs for HTML elements, so HTML tags in the input text will be removed first.

Usage

```
alnum_id(x, exclude = "[^[:alnum:]]+")
```

Arguments

`x`	A character vector.
`exclude`	A (Perl) regular expression to detect characters to be replaced by dashes. By default, non-alphanumeric characters are replaced.

Value

A character vector of IDs.

Examples

```
x = c("Hello world 123!", "a  &b*^##c 456")
xfun::alnum_id(x)
xfun::alnum_id(x, "[^[:alpha:]]+")  # only keep alphabetical chars
# when text contains HTML tags
xfun::alnum_id("<h1>Hello <strong>world</strong>!")
```

attr2 — *Obtain an attribute of an object without partial matching*

Description

An abbreviation of `base::attr(exact = TRUE)`.

Usage

```
attr2(...)
```

Arguments

`...`	Passed to `base::attr()` (without the `exact` argument).

Examples

```
z = structure(list(a = 1), foo = 2)
base::attr(z, "f")  # 2
xfun::attr2(z, "f")  # NULL
xfun::attr2(z, "foo")  # 2
```

base64_encode — *Encode/decode data into/from base64 encoding.*

Description

The function `base64_encode()` encodes a file or a raw vector into the base64 encoding. The function `base64_decode()` decodes data from the base64 encoding.

Usage

```
base64_encode(x)

base64_decode(x, from = NA)
```

Arguments

`x`	For `base64_encode()`, a raw vector. If not raw, it is assumed to be a file or a connection to be read via `readBin()`. For `base64_decode()`, a string.
`from`	If provided (and `x` is not provided), a connection or file to be read via `readChar()`, and the result will be passed to the argument `x`.

Value

`base64_encode()` returns a character string. `base64_decode()` returns a raw vector.

Examples

```
xfun::base64_encode(as.raw(1:10))
logo = xfun:::R_logo()
xfun::base64_encode(logo)
xfun::base64_decode("AQIDBAUGBwgJCg==")
```

---

base64_uri *Generate the Data URI for a file*

Description

Encode the file in the base64 encoding, and add the media type. The data URI can be used to embed data in HTML documents, e.g., in the `src` attribute of the `<img />` tag.

Usage

```
base64_uri(x, type = mime_type(x))
```

Arguments

`x`	A file path.
`type`	The MIME type of the file, e.g., `"image/png"` for a PNG image file.

Value

A string of the form `data:<media type>;base64,<data>`.

Examples

```
logo = xfun:::R_logo()
img = xfun::html_tag("img", src = xfun::base64_uri(logo), alt = "R logo")
if (interactive()) xfun::html_view(img)
```

---

base_pkgs *Get base R package names*

Description

Return base R package names.

Usage

```
base_pkgs()
```

Value

A character vector of base R package names.

Examples

```
xfun::base_pkgs()
```

`bg_process` *Start a background process*

Description

Start a background process using the PowerShell cmdlet `Start-Process-PassThru` on Windows or the ampersand `&` on Unix, and return the process ID.

Usage

```
bg_process(
  command,
  args = character(),
  verbose = getOption("xfun.bg_process.verbose", FALSE)
)
```

Arguments

`command, args`	The system command and its arguments. They do not need to be quoted, since they will be quoted via `shQuote()` internally.
`verbose`	If `FALSE`, suppress the output from `stdout` (and also `stderr` on Windows). The default value of this argument can be set via a global option, e.g., `options(xfun.bg_process.verbose = TRUE)`.

Value

The process ID as a character string.

Note

On Windows, if PowerShell is not available, try to use `system2(wait = FALSE)` to start the background process instead. The process ID will be identified from the output of the command `tasklist`. This method of looking for the process ID may not be reliable. If the search is not successful in 30 seconds, it will throw an error (timeout). If a longer time is needed, you may set `options(xfun.bg_process.timeout)` to a larger value, but it should be very rare that a process cannot be started in 30 seconds. When you reach the timeout, it is more likely that the command actually failed.

See Also

`proc_kill()` to kill a process.

`broken_packages` *Find out broken packages and reinstall them*

Description

If a package is broken (i.e., not `loadable()`), reinstall it.

Usage

```
broken_packages(reinstall = TRUE)
```

Arguments

`reinstall`	Whether to reinstall the broken packages, or only list their names.

Details

Installed R packages could be broken for several reasons. One common reason is that you have upgraded R to a newer `x.y` version, e.g., from `4.0.5` to `4.1.0`, in which case you need to reinstall previously installed packages.

Value

A character vector of names of broken package.

`browser_print` *Print a web page to PDF/PNG/JPEG*

Description

Print a web page to PDF or take a screenshot to PNG/JPEG via a headless browser such as Chromium or Google Chrome.

Usage

```
browser_print(
  input,
  output = ".pdf",
  args = "default",
  window_size = c(1280, 1024),
  browser = env_option("xfun.browser", find_browser())
)
```

Arguments

`input`	Path or URL to the HTML page to be printed.
`output`	An output filename. If only an extension is provided, the filename will be inferred from `url` via `url_filename()`. Only `.pdf`, `.png`, and `.jpeg` are supported.
`args`	Command-line arguments to be passed to the headless browser. The default arguments can be found in `xfun:::browser_args()`. You may pass additional arguments on top of these via, e.g., `c('default', '--no-pdf-header-footer')`, or completely override the default arguments by providing a vector of other arguments.
`window_size`	The browser window size when taking a PNG/JPEG screenshot. Ignored when printing to PDF.
`browser`	Path to the web browser. By default, the browser is found via `xfun:::find_browser()`. If it cannot be found, you may set the global option `options(xfun.browser = )` or environment variable `R_XFUN_BROWSER` to the path of the browser, or simply pass the path to the `browser` argument.

Value

The `output` path if the web page is successfully printed.

Examples

```
xfun::browser_print("https://www.r-project.org")
```

`bump_version` *Bump version numbers*

Description

Increase the last digit of version numbers, e.g., from `0.1` to `0.2`, or `7.23.9` to `7.23.10`.

Usage

```
bump_version(x)
```

Arguments

`x`	A vector of version numbers (of the class `"numeric_version"`), or values that can be coerced to version numbers via `as.numeric_version()`.

Value

A vector of new version numbers.

Examples

```
xfun::bump_version(c("0.1", "91.2.14"))
```

cache_exec — *Cache the execution of an expression in memory or on disk*

Description

Caching is based on the assumption that if the input does not change, the output will not change. After an expression is executed for the first time, its result will be saved (either in memory or on disk). The next run will be skipped and the previously saved result will be loaded directly if all external inputs of the expression remain the same, otherwise the cache will be invalidated and the expression will be re-executed.

Usage

```
cache_exec(expr, path = "cache/", id = NULL, ...)
```

Arguments

`expr`	An R expression to be cached.
`path`	The path to save the cache. The special value `":memory:"` means in-memory caching. If it is intended to be a directory path, please make sure to add a trailing slash.
`id`	A stable and unique string identifier for the expression to be used to identify a unique copy of cache for the current expression from all cache files (or in-memory elements). If not provided, an MD5 digest of the deparsed expression will be used, which means if the expression does not change (changes in comments or white spaces do not matter), the `id` will remain the same. This may not be a good default is two identical expressions are cached under the same `path`, because they could overwrite each other's cache when one expression's cache is invalidated, which may or may not be what you want. If you do not want that to happen, you need to manually provide an `id`.
`...`	More arguments to control the behavior of caching (see 'Details').

Details

Arguments supported in `...` include:

- `vars`: Names of local variables (which are created inside the expression). By default, local variables are automatically detected from the expression via `find_locals()`. Locally created variables are cached along with the value of the expression.
- `hash` and `extra`: R objects to be used to determine if cache should be loaded or invalidated. If (the MD5 hash of) the objects is not changed, the cache is loaded, otherwise the cache is invalidated and rebuilt. By default, `hash` is a list of values of global variables in the expression (i.e., variables created outside the expression). Global variables are automatically detected by

find_globals(). You can provide a vector of names to override the automatic detection if you want some specific global variables to affect caching, or the automatic detection is not reliable. You can also provide additional information via the extra argument. For example, if the expression reads an external file foo.csv, and you want the cache to be invalidated after the file is modified, you may use extra = file.mtime("foo.csv").

- keep: By default, only one copy of the cache corresponding to an id under path is kept, and all other copies for this id is automatically purged. If TRUE, all copies of the cache are kept. If FALSE, all copies are removed, which means the cache is *always* invalidated, and can be useful to force re-executing the expression.
- rw: A list of functions to read/write the cache files. The list is of the form list(name = 'xxx', load = function(file) {}, save = function(x, file) {}). By default, readRDS() and saveRDS() are used. This argument can also take a character string to use some built-in read/write methods. Currently available methods include rds (the default), raw (using serialize() and unserialize()), qs2 (using qs2::qs_read() and qs2::qs_save()), and qdata (using qs2::qd_read() and qs2::qd_save()). The rds and raw methods only use base R functions (the rds method generates smaller files because it uses compression, but is often slower than the raw method, which does not use compression). The qs2 and qdata methods require the **qs2** package, which can be much faster than base R methods and also supports compression.

Value

If the cache is found, the cached value of the expression will be loaded and returned (other local variables will also be lazy-loaded into the current environment as a side-effect). If cache does not exist, the expression is executed and its value is returned.

References

See https://yihui.org/litedown/#sec:option-cache for how it works and an application in **litedown**.

Examples

```
# the first run takes about 1 second
y1 = xfun::cache_exec({
    x = rnorm(1e+05)
    Sys.sleep(1)
    x
}, path = ":memory:", id = "sim-norm")

# the second run takes almost no time
y2 = xfun::cache_exec({
    # comments won't affect caching
    x = rnorm(1e+05)
    Sys.sleep(1)
    x
}, path = ":memory:", id = "sim-norm")

# y1, y2, and x should be identical
stopifnot(identical(y1, y2), identical(y1, x))
```

crandalf_check *Submit check jobs to crandalf*

Description

Check the reverse dependencies of a package using the crandalf service: `https://github.com/yihui/crandalf`. If the number of reverse dependencies is large, they will be split into batches and pushed to crandalf one by one.

Usage

```
crandalf_check(pkg, size = 400, jobs = Inf, which = "all")

crandalf_results(pkg, repo = NA, limit = 200, wait = 5 * 60)
```

Arguments

`pkg`	The package name of which the reverse dependencies are to be checked.
`size`	The number of reverse dependencies to be checked in each job.
`jobs`	The number of jobs to run in GitHub Actions (by default, all jobs are submitted, but you can choose to submit the first few jobs).
`which`	The type of dependencies (see `rev_check()`).
`repo`	The crandalf repo on GitHub (of the form `user/repo` such as `"yihui/crandalf"`). Usually you do not need to specify it, unless you are not calling this function inside the crandalf project, because `gh` should be able to figure out the repo automatically.
`limit`	The maximum of records for `gh run list` to retrieve. You only need a larger number if the check results are very early in the GitHub Action history.
`wait`	Number of seconds to wait if not all jobs have been completed on GitHub. By default, this function checks the status every 5 minutes until all jobs are completed. Set `wait` to 0 to disable waiting (and throw an error immediately when any jobs are not completed).

Details

Due to the time limit of a single job on GitHub Actions (6 hours), you will have to split the large number of reverse dependencies into batches and check them sequentially on GitHub (at most 5 jobs in parallel). The function `crandalf_check()` does this automatically when necessary. It requires the `git` command to be available.

The function `crandalf_results()` fetches check results from GitHub after all checks are completed, merge the results, and show a full summary of check results. It requires `gh` (GitHub CLI: `https://cli.github.com/manual/`) to be installed and you also need to authenticate with your GitHub account beforehand.

`csv_options` *Parse comma-separated chunk options*

Description

For **knitr** and R Markdown documents, code chunk options can be written using the comma-separated syntax (e.g., `opt1=value1, opt2=value2`). This function parses these options and returns a list. If an option is not named, it will be treated as the chunk label.

Usage

```
csv_options(x)
```

Arguments

`x`	The chunk options as a string.

Value

A list of chunk options.

Examples

```
xfun::csv_options("foo, eval=TRUE, fig.width=5, echo=if (TRUE) FALSE")
```

`decimal_dot` *Evaluate an expression after forcing the decimal point to be a dot*

Description

Sometimes it is necessary to use the dot character as the decimal separator. In R, this could be affected by two settings: the global option `options(OutDec)` and the `LC_NUMERIC` locale. This function sets the former to `.` and the latter to `C` before evaluating an expression, such as coercing a number to character.

Usage

```
decimal_dot(x)
```

Arguments

`x`	An expression.

Value

The value of `x`.

Examples

```
opts = options(OutDec = ",")
as.character(1.234)  # using ',' as the decimal separator
print(1.234)  # same
xfun::decimal_dot(as.character(1.234))  # using dot
xfun::decimal_dot(print(1.234))  # using dot
options(opts)
```

`del_empty_dir` *Delete an empty directory*

Description

Use `list.file()` to check if there are any files or subdirectories under a directory. If not, delete this empty directory.

Usage

```
del_empty_dir(dir)
```

Arguments

`dir`	Path to a directory. If `NULL` or the directory does not exist, no action will be performed.

`dir_create` *Create a directory recursively by default*

Description

First check if a directory exists. If it does, return `TRUE`, otherwise create it with `dir.create(recursive = TRUE)` by default.

Usage

```
dir_create(x, recursive = TRUE, ...)
```

Arguments

`x`	A path name.
`recursive`	Whether to create all directory components in the path.
`...`	Other arguments to be passed to `dir.create()`.

Value

A logical value indicating if the directory either exists or is successfully created.

`dir_exists` *Test the existence of files and directories*

Description

These are wrapper functions of `[utils::file_test()]` to test the existence of directories and files. Note that `file_exists()` only tests files but not directories, which is the main difference between `file.exists()` in base R. If you use are using the R version 3.2.0 or above, `dir_exists()` is the same as `dir.exists()` in base R.

Usage

```
dir_exists(x)

file_exists(x)
```

Arguments

`x`	A vector of paths.

Value

A logical vector.

`divide_chunk` *Divide chunk options from the code chunk body*

Description

Chunk options can be written in special comments (e.g., after `#|` for R code chunks) inside a code chunk. This function partitions these options from the chunk body.

Usage

```
divide_chunk(engine, code, strict = FALSE, ...)
```

Arguments

`engine`	The name of the language engine (to determine the appropriate comment character).
`code`	A character vector (lines of code).
`strict`	If `FALSE`, allow chunk options to be written after `#|` even if `#` is not the comment character of the engine (e.g., when `engine = 'js'`), otherwise throw an error if `#|` is detected but `#` is not the comment character.
`...`	Arguments to be passed to `yaml_load()`.

Value

A list with the following items:

- `options`: The parsed options (if there are any) as a list.
- `src`: The part of the input that contains the options.
- `code`: The part of the input that contains the code.

Note

Chunk options must be written on *continuous* lines (i.e., all lines must start with the special comment prefix such as `#|`) at the beginning of the chunk body.

Examples

```
# parse yaml-like items
yaml_like = c("#| label: mine", "#| echo: true", "#| fig.width: 8", "#| foo: bar",
    "1 + 1")
writeLines(yaml_like)
xfun::divide_chunk("r", yaml_like)

# parse CSV syntax
csv_like = c("#| mine, echo = TRUE, fig.width = 8, foo = 'bar'", "1 + 1")
writeLines(csv_like)
xfun::divide_chunk("r", csv_like)
```

`download_cache` *Download a file from a URL and cache it on disk*

Description

This object provides methods to download files and cache them on disk.

Usage

```
download_cache
```

Format

A list of methods:

- `$get(url, type, handler)` downloads a URL, caches it, and returns the file content according to the value of `type` (possible values: `"text"` means the text content; `"base64"` means the base64 encoded data; `"raw"` means the raw binary content; `"auto"` is the default and means the type is determined by the content type in the URL headers). Optionally a `handler` function can be applied to the content.
- `$list()` gives the list of cache files.
- `$summary()` gives a summary of existing cache files.
- `$remove(url, type)` removes a single cache file.
- `$purge()` deletes all cache files.

Examples

```
# the first time it may take a few seconds
x1 = xfun::download_cache$get("https://www.r-project.org/")
head(x1)

# now you can get the cached content
x2 = xfun::download_cache$get("https://www.r-project.org/")
identical(x1, x2)  # TRUE

# a binary file
x3 = xfun::download_cache$get("https://yihui.org/images/logo.png", "raw")
length(x3)

# show a summary
xfun::download_cache$summary()
# remove a specific cache file
xfun::download_cache$remove("https://yihui.org/images/logo.png", "raw")
# remove all cache files
xfun::download_cache$purge()
```

`download_file` *Try various methods to download a file*

Description

Try all possible methods in `download.file()` (e.g., `libcurl`, `curl`, `wget`, and `wininet`) and see if any method can succeed. The reason to enumerate all methods is that sometimes the default method does not work, e.g., `https://stat.ethz.ch/pipermail/r-devel/2016-June/072852.html`.

Usage

```
download_file(
  url,
  output = url_filename(url),
  ...,
  .error = "No download method works (auto/wininet/wget/curl/lynx)"
)
```

Arguments

`url`	The URL of the file.
`output`	Path to the output file. By default, it is determined by `url_filename()`.
`...`	Other arguments to be passed to `download.file()` (except `method`).
`.error`	An error message to signal when the download fails.

Value

The `output` file path if the download succeeded, or an error if none of the download methods worked.

Note

To allow downloading large files, the `timeout` option in `options()` will be temporarily set to one hour (3600 seconds) inside this function when this option has the default value of 60 seconds. If you want a different `timeout` value, you may set it via `options(timeout = N)`, where `N` is the number of seconds (not 60).

`do_once` *Perform a task once in an R session*

Description

Perform a task once in an R session, e.g., emit a message or warning. Then give users an optional hint on how not to perform this task at all.

Usage

```
do_once(
  task,
  option,
  hint = c("You will not see this message again in this R session.",
    "If you never want to see this message,",
    sprintf("you may set options(%s = FALSE) in your .Rprofile.", option))
)
```

Arguments

`task`	Any R code expression to be evaluated once to perform a task, e.g., `warning('Danger!')` or `message('Today is ', Sys.Date())`.
`option`	An R option name. This name should be as unique as possible in `options()`. After the task has been successfully performed, this option will be set to `FALSE` in the current R session, to prevent the task from being performed again the next time when `do_once()` is called.
`hint`	A character vector to provide a hint to users on how not to perform the task or see the message again in the current R session. Set `hint = ""` if you do not want to provide the hint.

Value

The value returned by the `task`, invisibly.

Examples

```
do_once(message("Today's date is ", Sys.Date()), "xfun.date.reminder")
# if you run it again, it will not emit the message again
do_once(message("Today's date is ", Sys.Date()), "xfun.date.reminder")

do_once({
    Sys.sleep(2)
    1 + 1
}, "xfun.task.1plus1")
do_once({
    Sys.sleep(2)
    1 + 1
}, "xfun.task.1plus1")
```

`embed_file` *Embed a file, multiple files, or directory on an HTML page*

Description

For a file, first encode it into base64 data (a character string). Then generate a hyperlink of the form '`<a href="base64 data" download="filename">Download filename</a>`'. The file can be downloaded when the link is clicked in modern web browsers. For a directory, it will be compressed as a zip archive first, and the zip file is passed to `embed_file()`. For multiple files, they are also compressed to a zip file first.

Usage

```
embed_file(path, name = basename(path), text = paste("Download", name), ...)

embed_dir(path, name = paste0(normalize_path(path), ".zip"), ...)

embed_files(path, name = with_ext(basename(path[1]), ".zip"), ...)
```

Arguments

`path`	Path to the file(s) or directory.
`name`	The default filename to use when downloading the file. Note that for `embed_dir()`, only the base name (of the zip filename) will be used.
`text`	The text for the hyperlink.
`...`	For `embed_file()`, additional arguments to be passed to `html_tag()` (e.g., `class = 'foo'`). For `embed_dir()` and `embed_files()`, arguments passed to `embed_file()`.

Details

These functions can be called in R code chunks in R Markdown documents with HTML output formats. You may embed an arbitrary file or directory in the HTML output file, so that readers of the HTML page can download it from the browser. A common use case is to embed data files for readers to download.

Value

An HTML tag '<a>' with the appropriate attributes.

Note

Windows users may need to install Rtools to obtain the `zip` command to use `embed_dir()` and `embed_files()`.

Internet Explorer does not support downloading embedded files. Chrome has a 2MB limit on the file size.

Examples

```
logo = xfun:::R_logo()
link = xfun::embed_file(logo, text = "Download R logo")
link
if (interactive()) xfun::html_view(link)
```

---

env_option — *Retrieve a global option from both* `options()` *and environment variables*

Description

If the option exists in `options()`, use its value. If not, query the environment variable with the name `R_NAME` where `NAME` is the capitalized option name with dots substituted by underscores. For example, for an option `xfun.foo`, first we try `getOption('xfun.foo')`; if it does not exist, we check the environment variable `R_XFUN_FOO`.

Usage

```
env_option(name, default = NULL)
```

Arguments

`name`	The option name.
`default`	The default value if the option is not found in `options()` or environment variables.

Details

This provides two possible ways, whichever is more convenient, for users to set an option. For example, global options can be set in the .Rprofile file, and environment variables can be set in the .Renviron file.

Value

The option value.

Examples

```
xfun::env_option("xfun.test.option")  # NULL

Sys.setenv(R_XFUN_TEST_OPTION = "1234")
xfun::env_option("xfun.test.option")  # 1234

options(xfun.test.option = TRUE)
xfun::env_option("xfun.test.option")  # TRUE (from options())
options(xfun.test.option = NULL)  # reset the option
xfun::env_option("xfun.test.option")  # 1234 (from env var)

Sys.unsetenv("R_XFUN_TEST_OPTION")
xfun::env_option("xfun.test.option")  # NULL again

xfun::env_option("xfun.test.option", FALSE)  # use default
```

`existing_files` *Find file paths that exist*

Description

This is a shorthand of `x[file.exists(x)]`, and optionally returns the first existing file path.

Usage

```
existing_files(x, first = FALSE, error = TRUE)
```

Arguments

`x`	A vector of file paths.
`first`	Whether to return the first existing path. If `TRUE` and no specified files exist, it will signal an error unless the argument `error = FALSE`.
`error`	Whether to throw an error when `first = TRUE` but no files exist. It can also take a character value, which will be used as the error message.

Value

A vector of existing file paths.

Examples

```
xfun::existing_files(c("foo.txt", system.file("DESCRIPTION", package = "xfun")))
```

exit_call *Call* on.exit() *in a parent function*

Description

The function on.exit() is often used to perform tasks when the current function exits. This exit_call() function allows calling a function when a parent function exits (thinking of it as inserting an on.exit() call into the parent function).

Usage

```
exit_call(fun, n = 2, ...)
```

Arguments

fun	A function to be called when the parent function exits.
n	The parent frame number. For n = 1, exit_call(fun) is the same as on.exit(fun()); n = 2 means adding on.exit(fun()) in the parent function; n = 3 means the grandparent, etc.
...	Other arguments to be passed to on.exit().

References

This function was inspired by Kevin Ushey: https://yihui.org/en/2017/12/on-exit-parent/

Examples

```
f = function(x) {
    print(x)
    xfun::exit_call(function() print("The parent function is exiting!"))
}
g = function(y) {
    f(y)
    print("f() has been called!")
}
g("An argument of g()!")
```

fenced_block *Create a fenced block in Markdown*

Description

Wrap content with fence delimiters such as backticks (code blocks) or colons (fenced Div). Optionally the fenced block can have attributes. The function fenced_div() is a shorthand of fenced_block(char = ':').

Usage

```
fenced_block(x, attrs = NULL, fence = make_fence(x, char), char = "`")

fenced_div(...)

make_fence(x, char = "`", start = 3)
```

Arguments

`x`	A character vector of the block content.
`attrs`	A vector of block attributes.
`fence`	The fence string, e.g., `:::` or ```` ``` ````. This will be generated from the `char` argument by default.
`char`	The fence character to be used to generate the fence string by default.
`...`	Arguments to be passed to `fenced_block()`.
`start`	The number of characters to start searching `x` with. If the string of this number of characters is found, add one more character, and repeat the search.

Value

`fenced_block()` returns a character vector that contains both the fences and content.

`make_fence()` returns a character string. If the block content contains `N` fence characters (e.g., backticks), use `N + 1` characters as the fence.

Examples

`````
# code block with class 'r' and ID 'foo'
xfun::fenced_block("1+1", c(".r", "#foo"))
# fenced Div
xfun::fenced_block("This is a **Div**.", char = ":")
# three backticks by default
xfun::make_fence("1+1")
# needs five backticks for the fences because content has four
xfun::make_fence(c("````r", "1+1", "````"))
`````

`file_ext` *Manipulate filename extensions*

Description

Functions to obtain (`file_ext()`), remove (`sans_ext()`), and change (`with_ext()`) extensions in filenames.

Usage

```
file_ext(x, extra = "")

sans_ext(x, extra = "")

with_ext(x, ext, extra = "")
```

Arguments

x	A character of file paths.
extra	Extra characters to be allowed in the extensions. By default, only alphanumeric characters are allowed (and also some special cases in 'Details'). If other characters should be allowed, they can be specified in a character string, e.g., `"-+!_#"`.
ext	A vector of new extensions. It must be either of length 1, or the same length as `x`.

Details

`file_ext()` is similar to `tools::file_ext()`, and `sans_ext()` is similar to `tools::file_path_sans_ext()`. The main differences are that they treat `tar.(gz|bz2|xz)` and `nb.html` as extensions (but functions in the **tools** package doesn't allow double extensions by default), and allow characters ~ and # to be present at the end of a filename.

Value

A character vector of the same length as `x`.

Examples

```
library(xfun)
p = c("abc.doc", "def123.tex", "path/to/foo.Rmd", "backup.ppt~", "pkg.tar.xz")
file_ext(p)
sans_ext(p)
with_ext(p, ".txt")
with_ext(p, c(".ppt", ".sty", ".Rnw", "doc", "zip"))
with_ext(p, "html")

# allow for more characters in extensions
p = c("a.c++", "b.c--", "c.e##")
file_ext(p)  # -/+/# not recognized by default
file_ext(p, extra = "-+#")
```

`file_rename` *Rename files and directories*

Description

First try `file.rename()`. If it fails (e.g., renaming a file from one volume to another on disk is likely to fail), try `file.copy()` instead, and clean up the original files if the copy succeeds.

Usage

```
file_rename(from, to)
```

Arguments

`from, to`	Original and target paths, respectively.

Value

A logical vector (`TRUE` for success and `FALSE` for failure).

`file_string` *Read a text file and concatenate the lines by* `'\n'`

Description

The source code of this function should be self-explanatory.

Usage

```
file_string(file)
```

Arguments

`file`	Path to a text file (should be encoded in UTF-8).

Value

A character string of text lines concatenated by `'\n'`.

Examples

```
xfun::file_string(system.file("DESCRIPTION", package = "xfun"))
```

`find_globals` *Find global/local variables in R code*

Description

Use `codetools::findGlobals()` and `codetools::findLocalsList()` to find global and local variables in a piece of code. Global variables are defined outside the code, and local variables are created inside the code.

Usage

```
find_globals(code, envir = parent.frame())

find_locals(code)
```

Arguments

`code`	Either a character vector of R source code, or an R expression.
`envir`	The global environment in which global variables are to be found.

Value

A character vector of the variable names. If the source code contains syntax errors, an empty character vector will be returned.

Note

Due to the flexibility of creating and getting variables in R, these functions are not guaranteed to find all possible variables in the code (e.g., when the code is hidden behind `eval()`).

Examples

```
x = 2
xfun::find_globals("y = x + 1")
xfun::find_globals("y = get('x') + 1")  # x is not recognized
xfun::find_globals("y = zzz + 1")  # zzz doesn't exist

xfun::find_locals("y = x + 1")
xfun::find_locals("assign('y', x + 1)")  # it works
xfun::find_locals("assign('y', x + 1, new.env())")  # still smart
xfun::find_locals("eval(parse(text = 'y = x + 1'))")  # no way
```

format_bytes — *Format numbers of bytes using a specified unit*

Description

Call the S3 method `format.object_size()` to format numbers of bytes.

Usage

```
format_bytes(x, units = "auto", ...)
```

Arguments

`x`	A numeric vector (each element represents a number of bytes).
`units, ...`	Passed to `format()`.

Value

A character vector.

Examples

```
xfun::format_bytes(c(1, 1024, 2000, 1e+06, 2e+08))
xfun::format_bytes(c(1, 1024, 2000, 1e+06, 2e+08), units = "KB")
```

from_root — *Get the relative path of a path in a project relative to the current working directory*

Description

First compose an absolute path using the project root directory and the relative path components, i.e., `file.path(root, ...)`. Then convert it to a relative path with `relative_path()`, which is relative to the current working directory.

Usage

```
from_root(..., root = proj_root(), error = TRUE)
```

Arguments

`...`	A character vector of path components *relative to the root directory of the project*.
`root`	The root directory of the project.
`error`	Whether to signal an error if the path cannot be converted to a relative path.

Details

This function was inspired by `here::here()`, and the major difference is that it returns a relative path by default, which is more portable.

Value

A relative path, or an error when the project root directory cannot be determined or the conversion failed and `error = TRUE`.

Examples

```
## Not run:
xfun::from_root("data", "mtcars.csv")

## End(Not run)
```

`github_releases` *Get the tags of GitHub releases of a repository*

Description

Use the GitHub API (`github_api()`) to obtain the tags of the releases.

Usage

```
github_releases(
  repo,
  tag = "",
  pattern = "v[0-9.]+",
  use_jsonlite = loadable("jsonlite")
)
```

Arguments

`repo`	The repository name of the form `user/repo`, e.g., `"yihui/xfun"`.
`tag`	A tag as a character string. If provided, it will be returned if the tag exists. If `tag = "latest"`, the tag of the latest release is returned.
`pattern`	A regular expression to match the tags.
`use_jsonlite`	Whether to use **jsonlite** to parse the releases info.

Value

A character vector of (GIT) tags.

Examples

```
xfun::github_releases("yihui/litedown")
xfun::github_releases("gohugoio/hugo", "latest")
```

grep_sub *Perform replacement with* `gsub()` *on elements matched from* `grep()`

Description

This function is a shorthand of `gsub(pattern, replacement, grep(pattern, x, value = TRUE))`.

Usage

```
grep_sub(pattern, replacement, x, ...)
```

Arguments

`pattern, replacement, x, ...`
: Passed to `grep()` and `gsub()`.

Value

A character vector.

Examples

```
# find elements that matches 'a[b]+c' and capitalize 'b' with perl regex
xfun::grep_sub("a([b]+)c", "a\\U\\1c", c("abc", "abbbc", "addc", "123"), perl = TRUE)
```

gsub_file *Search and replace strings in files*

Description

These functions provide the "file" version of `gsub()`, i.e., they perform searching and replacement in files via `gsub()`.

Usage

```
gsub_file(file, ..., rw_error = TRUE)

gsub_files(files, ...)

gsub_dir(..., dir = ".", recursive = TRUE, ext = NULL, mimetype = ".*")

gsub_ext(ext, ..., dir = ".", recursive = TRUE)
```

Arguments

`file`	Path of a single file.
`...`	For `gsub_file()`, arguments passed to `gsub()`. For other functions, arguments passed to `gsub_file()`. Note that the argument `x` of `gsub()` is the content of the file.
`rw_error`	Whether to signal an error if the file cannot be read or written. If `FALSE`, the file will be ignored (with a warning).
`files`	A vector of file paths.
`dir`	Path to a directory (all files under this directory will be replaced).
`recursive`	Whether to find files recursively under a directory.
`ext`	A vector of filename extensions (without the leading periods).
`mimetype`	A regular expression to filter files based on their MIME types, e.g., `'^text/'` for plain text files.

Note

These functions perform in-place replacement, i.e., the files will be overwritten. Make sure you backup your files in advance, or use version control!

Examples

```
library(xfun)
f = tempfile()
writeLines(c("hello", "world"), f)
gsub_file(f, "world", "woRld", fixed = TRUE)
readLines(f)
```

`html_tag` *Tools for HTML tags*

Description

Given a tag name, generate an HTML tag with optional attributes and content. `html_tag()` can be viewed as a simplified version of `htmltools::tags`, `html_value()` adds classes on the value so that it will be treated as raw HTML (not escaped by `html_tag()`), `html_escape()` escapes special characters in HTML, and `html_view()` launches a browser or viewer to view the HTML content.

Usage

```
html_tag(.name, .content = NULL, .attrs = NULL, ...)

html_value(x)

html_escape(x, attr = FALSE)

html_view(x, ...)
```

Arguments

`.name`	The tag name.
`.content`	The content between opening and closing tags. Ignored for void tags such as `<img>`. Special characters such as &, <, and > will be escaped unless the value was generated from `html_value()`. The content can be either a character vector or a list. If it is a list, it may contain both normal text and HTML content.
`.attrs`	A named list of attributes.
`...`	For `html_tag()`, named arguments as an alternative way to provide attributes. For `html_view()`, other arguments to be passed to `new_app()`.
`x`	A character vector to be treated as raw HTML content for `html_value()`, escaped for `html_escape()`, and viewed for `html_view()`.
`attr`	Whether to escape ", `\r`, and `\n` (which should be escaped for tag attributes).

Value

A character string.

Examples

```
xfun::html_tag("a", "<R Project>", href = "https://www.r-project.org", target = "_blank")
xfun::html_tag("br")
xfun::html_tag("a", xfun::html_tag("strong", "R Project"), href = "#")
xfun::html_tag("a", list("<text>", xfun::html_tag("b", "R Project")), href = "#")
xfun::html_escape("\" quotes \" & brackets < >")
xfun::html_escape("\" & < > \r \n", attr = TRUE)
```

`install_dir` *Install a source package from a directory*

Description

Run `R CMD build` to build a tarball from a source directory, and run `R CMD INSTALL` to install it.

Usage

```
install_dir(pkg = ".", build = TRUE, build_opts = NULL, install_opts = NULL)
```

Arguments

`pkg`	The package source directory.
`build`	Whether to build a tarball from the source directory. If `FALSE`, run `R CMD INSTALL` on the directory directly (note that vignettes will not be automatically built).
`build_opts`	The options for `R CMD build`.
`install_opts`	The options for `R CMD INSTALL`.

Value

Invisible status from `R CMD INSTALL`.

install_github — *An alias of* remotes::install_github()

Description

This alias is to make autocomplete faster via xfun::install_github, because most remotes::install_* functions are never what I want. I only use install_github and it is inconvenient to autocomplete it, e.g. install_git always comes before install_github, but I never use it. In RStudio, I only need to type xfun::ig to get xfun::install_github.

Usage

```
install_github(...)
```

Arguments

...	Arguments to be passed to remotes::install_github().

in_dir — *Evaluate an expression under a specified working directory*

Description

Change the working directory, evaluate the expression, and restore the working directory.

Usage

```
in_dir(dir, expr)
```

Arguments

dir	Path to a directory.
expr	An R expression.

Examples

```
library(xfun)
in_dir(tempdir(), {
    print(getwd())
    list.files()
})
```

`is_abs_path` *Test if paths are relative or absolute*

Description

On Unix, check if the paths start with '`/`' or '`~`' (if they do, they are absolute paths). On Windows, check if a path remains the same (via `same_path()`) if it is prepended with '`./`' (if it does, it is a relative path).

Usage

```
is_abs_path(x)

is_rel_path(x)
```

Arguments

`x`	A vector of paths.

Value

A logical vector.

Examples

```
xfun::is_abs_path(c("C:/foo", "foo.txt", "/Users/john/", tempdir()))
xfun::is_rel_path(c("C:/foo", "foo.txt", "/Users/john/", tempdir()))
```

`is_ascii` *Check if a character vector consists of entirely ASCII characters*

Description

Converts the encoding of a character vector to `'ascii'`, and check if the result is `NA`.

Usage

```
is_ascii(x)
```

Arguments

`x`	A character vector.

Value

A logical vector indicating whether each element of the character vector is ASCII.

Examples

```
library(xfun)
is_ascii(letters)  # yes
is_ascii(intToUtf8(8212))  # no
```

`is_blank` *Test if a character vector consists of blank strings*

Description

Return a logical vector indicating if elements of a character vector are blank (white spaces or empty strings).

Usage

```
is_blank(x)
```

Arguments

`x`	A character vector.

Value

`TRUE` for blank elements, or `FALSE` otherwise.

Examples

```
xfun::is_blank("")
xfun::is_blank("abc")
xfun::is_blank(c("", "  ", "\n\t"))
xfun::is_blank(c("", " ", "abc"))
```

`is_sub_path` *Test if a path is a subpath of a dir*

Description

Check if the path starts with the dir path.

Usage

```
is_sub_path(x, dir, n = nchar(dir))
```

Arguments

`x`	A vector of paths.
`dir`	A vector of directory paths.
`n`	The length of `dir` paths.

Value

A logical vector.

Note

You may want to normalize the values of the `x` and `dir` arguments first (with `normalize_path()`), to make sure the path separators are consistent.

Examples

```
xfun::is_sub_path("a/b/c.txt", "a/b")  # TRUE
xfun::is_sub_path("a/b/c.txt", "d/b")  # FALSE
xfun::is_sub_path("a/b/c.txt", "a\\b")  # FALSE (even on Windows)
```

`is_web_path` *Test if a path is a web path*

Description

Check if a path starts with ‘`http://`’ or ‘`https://`’ or ‘`ftp://`’ or ‘`ftps://`’.

Usage

```
is_web_path(x)
```

Arguments

`x`	A vector of paths.

Value

A logical vector.

Examples

```
xfun::is_web_path("https://www.r-project.org")  # TRUE
xfun::is_web_path("www.r-project.org")  # FALSE
```

is_windows *Test for types of operating systems*

Description

Functions based on `.Platform$OS.type` and `Sys.info()` to test if the current operating system is Windows, macOS, Unix, or Linux.

Usage

```
is_windows()

is_unix()

is_macos()

is_linux()

is_arm64()
```

Examples

```
library(xfun)
# only one of the following statements should be true
is_windows()
is_unix() && is_macos()
is_linux()
# In newer Macs, CPU can be either Intel or Apple
is_arm64()  # TRUE on Apple silicone machines
```

join_words *Join multiple words into a single string*

Description

If `words` is of length 2, the first word and second word are joined by the `and` string; if `and` is blank, `sep` is used. When the length is greater than 2, `sep` is used to separate all words, and the `and` string is prepended to the last word.

Usage

```
join_words(
  words,
  sep = ", ",
  and = " and ",
  before = "",
  after = before,
  oxford_comma = TRUE
)
```

Arguments

`words`	A character vector.
`sep`	Separator to be inserted between words.
`and`	Character string to be prepended to the last word.
`before, after`	A character string to be added before/after each word.
`oxford_comma`	Whether to insert the separator between the last two elements in the list.

Value

A character string marked by `raw_string()`.

Examples

```
join_words("a")
join_words(c("a", "b"))
join_words(c("a", "b", "c"))
join_words(c("a", "b", "c"), sep = " / ", and = "")
join_words(c("a", "b", "c"), and = "")
join_words(c("a", "b", "c"), before = "\"", after = "\"")
join_words(c("a", "b", "c"), before = "\"", after = "\"", oxford_comma = FALSE)
```

`lazy_save` *Save objects to files and lazy-load them*

Description

The function `lazy_save()` saves objects to files with incremental integer names (e.g., the first object is saved to `1.rds`, and the second object is saved to `2.rds`, etc.). The function `lazy_load()` lazy-load objects from files saved via `lazy_save()`, i.e., a file will not be read until the object is used.

Usage

```
lazy_save(list = NULL, path = "./", method = "auto", envir = parent.frame())

lazy_load(path = "./", method = "auto", envir = parent.frame())
```

Arguments

`list`	A character vector of object names. This list will be written to an index file with `0` as the base name (e.g., `0.rds`).
`path`	The path to write files to / read files from.
`method`	The file save/load method. It can be a string (e.g., `rds`, `raw`, or `qs2`) or a list. See the `rw` argument of `cache_exec()`. By default, it is automatically detected by checking the existence of the index file (e.g., `0.rds`, `0.raw`, or `0.qs2`).
`envir`	The environment to get or assign objects.

Value

`lazy_save()` returns invisible `NULL`; `lazy_load()` returns the object names invisibly.

See Also

`delayedAssign()`

---

magic_path *Find a file or directory under a root directory*

Description

Given a path, try to find it recursively under a root directory. The input path can be an incomplete path, e.g., it can be a base filename, and `magic_path()` will try to find this file under subdirectories.

Usage

```
magic_path(
  ...,
  root = proj_root(),
  relative = TRUE,
  error = TRUE,
  message = getOption("xfun.magic_path.message", TRUE),
  n_dirs = getOption("xfun.magic_path.n_dirs", 10000)
)
```

Arguments

`...`	A character vector of path components.
`root`	The root directory under which to search for the path. If `NULL`, the current working directory is used.
`relative`	Whether to return a relative path.
`error`	Whether to signal an error if the path is not found, or multiple paths are found.
`message`	Whether to emit a message when multiple paths are found and `error = FALSE`.

`n_dirs`	The number of subdirectories to recursively search. The recursive search may be time-consuming when there are a large number of subdirectories under the root directory. If you really want to search for all subdirectories, you may try `n_dirs = Inf`.

Value

The path found under the root directory, or an error when `error = TRUE` and the path is not found (or multiple paths are found).

Examples

```
## Not run:
xfun::magic_path("mtcars.csv")  # find any file that has the base name mtcars.csv

## End(Not run)
```

`mark_dirs` *Mark some paths as directories*

Description

Add a trailing backlash to a file path if this is a directory. This is useful in messages to the console for example to quickly identify directories from files.

Usage

```
mark_dirs(x)
```

Arguments

`x`	Character vector of paths to files and directories.

Details

If `x` is a vector of relative paths, directory test is done with path relative to the current working dir. Use `in_dir()` or use absolute paths.

Examples

```
mark_dirs(list.files(find.package("xfun"), full.names = TRUE))
```

md5 *Calculate the MD5 checksums of R objects*

Description

Serialize an object and calculate the checksum via `tools::md5sum()`. If `tools::md5sum()` does not have the argument `bytes`, the object will be first serialized to a temporary file, which will be deleted after the checksum is calculated, otherwise the raw bytes of the object will be passed to the `bytes` argument directly (which will be faster than writing to a temporary file).

Usage

```
md5(...)
```

Arguments

`...`	Any number of R objects.

Value

A character vector of the checksums of objects passed to `md5()`. If the arguments are named, the results will also be named.

Examples

```
x1 = 1
x2 = 1:10
x3 = seq(1, 10)
x4 = iris
x5 = paste
(m = xfun::md5(x1, x2, x3, x4, x5))
stopifnot(m[2] == m[3])  # x2 and x3 should be identical

xfun::md5(x1 = x1, x2 = x2)  # named arguments
```

md_table *Generate a simple Markdown pipe table*

Description

A minimal Markdown table generator using the pipe `|` as column separators.

Usage

```
md_table(x, digits = NULL, na = NULL, newline = NULL, limit = NULL)
```

Arguments

`x`	A 2-dimensional object (e.g., a matrix or data frame).
`digits`	The number of decimal places to be passed to `round()`. It can be a integer vector of the same length as the number of columns in `x` to round columns separately. The default is `3`.
`na`	A character string to represent `NA` values. The default is an empty string.
`newline`	A character string to substitute `\n` in `x` (because pipe tables do not support line breaks in cells). The default is a space.
`limit`	The maximum number of rows to show in the table. If it is smaller than the number of rows, the data in the middle will be omitted. If it is of length 2, the second number will be used to limit the number of columns. Zero and negative values are ignored.

Details

The default argument values can be set via global options with the prefix `xfun.md_table.`, e.g., `options(xfun.md_table.digits 2, xfun.md_table.na = 'n/a')`.

Value

A character vector.

See Also

`knitr::kable()` (which supports more features)

Examples

```
xfun::md_table(head(iris))
xfun::md_table(mtcars, limit = c(10, 6))
```

`mime_type` *Get the MIME types of files*

Description

If the **mime** package is installed, call `mime::guess_type()`, otherwise use the system command `file --mime-type` to obtain the MIME type of a file. Typically, the `file` command exists on *nix. On Windows, the command should exist if Cygwin or Rtools is installed. If it is not found, .NET's `MimeMapping` class will be used instead (which requires the .NET framework).

Usage

```
mime_type(x, use_mime = loadable("mime"), empty = "text/plain")
```

Arguments

`x`	A vector of file paths.
`use_mime`	Whether to use the **mime** package.
`empty`	The MIME type for files without extensions (e.g., `Makefile`). If `NA`, the type will be obtained from system command. This argument is used only for `use_mime = FALSE`.

Value

A character vector of MIME types.

Note

When querying the MIME type via the system command, the result will be cached to `xfun:::cache_dir()`. This will make future queries much faster, since running the command in real time can be a little slow.

Examples

```
f = list.files(R.home("doc"), full.names = TRUE)
mime_type(f)
mime_type(f, FALSE)  # don't use mime
mime_type(f, FALSE, NA)  # run command for files without extension
```

`msg_cat` *Generate a message with* `cat()`

Description

This function is similar to `message()`, and the difference is that `msg_cat()` uses `cat()` to write out the message, which is sent to `stdout()` instead of `stderr()`. The message can be suppressed by `suppressMessages()`.

Usage

```
msg_cat(...)
```

Arguments

`...`	Character strings of messages, which will be concatenated into one string via `paste(c(...), collapse = '')`.

Value

Invisible `NULL`, with the side-effect of printing the message.

Note

By default, a newline will not be appended to the message. If you need a newline, you have to explicitly add it to the message (see 'Examples').

See Also

This function was inspired by `rlang::inform()`.

Examples

```
{
    # a message without a newline at the end
    xfun::msg_cat("Hello world!")
    # add a newline at the end
    xfun::msg_cat(" This message appears right after the previous one.\n")
}
suppressMessages(xfun::msg_cat("Hello world!"))
```

`native_encode` *Try to use the system native encoding to represent a character vector*

Description

Apply `enc2native()` to the character vector, and check if `enc2utf8()` can convert it back without a loss. If it does, return `enc2native(x)`, otherwise return the original vector with a warning.

Usage

```
native_encode(x)
```

Arguments

`x`	A character vector.

Note

On platforms that supports UTF-8 as the native encoding (`l10n_info()[['UTF-8']]` returns `TRUE`), the conversion will be skipped.

Examples

```
library(xfun)
s = intToUtf8(c(20320, 22909))
Encoding(s)

s2 = native_encode(s)
Encoding(s2)
```

news2md — *Convert package news to the Markdown format*

Description

Read the package news with `news()`, convert the result to Markdown, and write to an output file (e.g., ‘`NEWS.md`’). Each package version appears in a first-level header, each category (e.g., ‘`NEW FEATURES`’ or ‘`BUG FIXES`’) is in a second-level header, and the news items are written into bullet lists.

Usage

```
news2md(package, ..., output = "NEWS.md", category = TRUE)
```

Arguments

`package, ...`	Arguments to be passed to `news()`.
`output`	The output file path.
`category`	Whether to keep the category names.

Value

If `output = NA`, returns the Markdown content as a character vector, otherwise the content is written to the output file.

Examples

```
# news for the current version of R
xfun::news2md("R", Version == getRversion(), output = NA)
```

new_app — *Create a local web application*

Description

An experimental function to create a local web application based on R’s internal `httpd` server (which is primarily for running R’s dynamic help system).

Usage

```
new_app(name, handler, open = interactive(), ports = 4321 + 1:10)
```

Arguments

`name`	The app name (a character string, and each app should have a unique name).
`handler`	A function that takes the HTTP request information (the first argument is the requested path) and returns a response.
`open`	Whether to open the app, or a function to open the app URL.
`ports`	A vector of ports to try for starting the server.

Value

The app URL of the form `http://127.0.0.1:port/custom/name/`.

Note

This function is not based on base R's public API, and is possible to break in the future, which is also why the documentation here is terse. Please avoid creating public-facing web apps with it. You may consider packages like **httpuv** and **Rserve** for production web apps.

`normalize_path` *Normalize paths*

Description

A wrapper function of `normalizePath()` with different defaults.

Usage

```
normalize_path(x, winslash = "/", must_work = FALSE, resolve_symlink = TRUE)
```

Arguments

`x`, `winslash`, `must_work`	Arguments passed to `normalizePath()`.
`resolve_symlink`	Whether to resolve symbolic links.

Examples

```
library(xfun)
normalize_path("~")
```

`numbers_to_words` *Convert numbers to English words*

Description

This can be helpful when writing reports with **knitr/rmarkdown** if we want to print numbers as English words in the output. The function `n2w()` is an alias of `numbers_to_words()`.

Usage

```
numbers_to_words(x, cap = FALSE, hyphen = TRUE, and = FALSE)

n2w(x, cap = FALSE, hyphen = TRUE, and = FALSE)
```

Arguments

`x`	A numeric vector. The absolute values should be less than `1e15`.
`cap`	Whether to capitalize the first letter of the word. This can be useful when the word is at the beginning of a sentence. Default is `FALSE`.
`hyphen`	Whether to insert hyphen (-) when the number is between 21 and 99 (except 30, 40, etc.).
`and`	Whether to insert `and` between hundreds and tens, e.g., write 110 as “one hundred and ten” if `TRUE` instead of “one hundred ten”.

Value

A character vector.

Author(s)

Daijiang Li

Examples

```
library(xfun)
n2w(0, cap = TRUE)
n2w(0:121, and = TRUE)
n2w(1e+06)
n2w(1e+11 + 12345678)
n2w(-987654321)
n2w(1e+15 - 1)
n2w(123.456)
n2w(123.45678901)
n2w(123.456789098765)
```

`optipng` *Run OptiPNG on all PNG files under a directory*

Description

Call the command `optipng` via `system2()` to optimize all PNG files under a directory.

Usage

```
optipng(dir = ".", files = all_files("[.]png$", dir), ...)
```

Arguments

`dir`	Path to a directory.
`files`	Alternatively, you can choose the specific files to optimize.
`...`	Arguments to be passed to `system2()`.

References

OptiPNG: `https://optipng.sourceforge.net`.

`parse_only` *Parse R code and do not keep the source*

Description

An abbreviation of `parse(keep.source = FALSE)`.

Usage

```
parse_only(code)
```

Arguments

`code`	A character vector of the R source code.

Value

R `expression()`s.

Examples

```
library(xfun)
parse_only("1+1")
parse_only(c("y~x", "1:5 # a comment"))
parse_only(character(0))
```

pkg_attach — *Attach or load packages, and automatically install missing packages if requested*

Description

pkg_attach() is a vectorized version of library() over the package argument to attach multiple packages in a single function call. pkg_load() is a vectorized version of requireNamespace() to load packages (without attaching them). The functions pkg_attach2() and pkg_load2() are wrappers of pkg_attach(install = TRUE) and pkg_load(install = TRUE), respectively. loadable() is an abbreviation of requireNamespace(quietly = TRUE). pkg_available() tests if a package with a minimal version is available.

Usage

```
pkg_attach(
  ...,
  install = FALSE,
  message = getOption("xfun.pkg_attach.message", TRUE)
)

pkg_load(..., error = TRUE, install = FALSE)

loadable(pkg, strict = TRUE, new_session = FALSE)

pkg_available(pkg, version = NULL)

pkg_attach2(...)

pkg_load2(...)
```

Arguments

...	Package names (character vectors, and must always be quoted).
install	Whether to automatically install packages that are not available using install.packages(). Besides TRUE and FALSE, the value of this argument can also be a function to install packages (install = TRUE is equivalent to install = install.packages), or a character string "pak" (equivalent to install = pak::pkg_install, which requires the **pak** package). You are recommended to set a CRAN mirror in the global option repos via options() if you want to automatically install packages.
message	Whether to show the package startup messages (if any startup messages are provided in a package).
error	Whether to signal an error when certain packages cannot be loaded.
pkg	A single package name.

`strict`	If `TRUE`, use `requireNamespace()` to test if a package is loadable; otherwise only check if the package is in `.packages(TRUE)` (this does not really load the package, so it is less rigorous but on the other hand, it can keep the current R session clean).
`new_session`	Whether to test if a package is loadable in a new R session. Note that `new_session = TRUE` implies `strict = TRUE`.
`version`	A minimal version number. If `NULL`, only test if a package is available and do not check its version.

Details

These are convenience functions that aim to solve these common problems: (1) We often need to attach or load multiple packages, and it is tedious to type several `library()` calls; (2) We are likely to want to install the packages when attaching/loading them but they have not been installed.

Value

`pkg_attach()` returns `NULL` invisibly. `pkg_load()` returns a logical vector, indicating whether the packages can be loaded.

See Also

`pkg_attach2()` is similar to `pacman::p_load()`, but does not allow non-standard evaluation (NSE) of the `...` argument, i.e., you must pass a real character vector of package names to it, and all names must be quoted. Allowing NSE adds too much complexity with too little gain (the only gain is that it saves your effort in typing two quotes).

Examples

```
library(xfun)
pkg_attach("stats", "graphics")
# pkg_attach2('servr') # automatically install servr if it is not installed

(pkg_load("stats", "graphics"))
```

---

pkg_bib — *Generate BibTeX bibliography databases for R packages*

Description

Call `utils::citation()` and `utils::toBibtex()` to create bib entries for R packages and write them in a file. It can facilitate the auto-generation of bibliography databases for R packages, and it is easy to regenerate all the citations after updating R packages.

Usage

```
pkg_bib(
  x = .packages(),
  file = "",
  tweak = TRUE,
  width = NULL,
  prefix = getOption("xfun.bib.prefix", "R-"),
  lib.loc = NULL,
  packageURL = TRUE,
  date = c("none", "package", "today")
)
```

Arguments

`x`	Package names. Packages which are not installed are ignored.
`file`	The (‘`.bib`’) file to write. By default, or if `NULL`, output is written to the R console.
`tweak`	Whether to fix some known problems in the citations, especially non-standard format of author names.
`width`	Width of lines in bibliography entries. If `NULL`, lines will not be wrapped.
`prefix`	Prefix string for keys in BibTeX entries; by default, it is `R-` unless `option('xfun.bib.prefix')` has been set to another string.
`lib.loc`	A vector of path names of R libraries.
`packageURL`	Use the `URL` field from the ‘`DESCRIPTION`’ file. See Details below.
`date`	The way to include a date of the form `(retrieved YYYY-mm-dd)` in the `note` field of bib entries. For `date = 'none'`, no date is included. For `date = 'package'`, the `packageDate()` of the package is included. For `date = 'today'`, `Sys.Date()` is included. Alternatively, you can provide a date directly, e.g., `date = '2025-04-01'`.

Details

For a package, the keyword `R-pkgname` is used for its bib item, where `pkgname` is the name of the package. Citation entries specified in the ‘`CITATION`’ file of the package are also included. The main purpose of this function is to automate the generation of the package citation information because it often changes (e.g., the author, year, package version, and so on).

There are at least two different uses for the URL in a reference list. You might want to tell users where to go for more information. In that case, use the default `packageURL = TRUE`, and the first URL listed in the ‘`DESCRIPTION`’ file will be used. Be careful: some authors don’t put the most relevant URL first. Alternatively, you might want to identify exactly which version of the package was used in the document. If it was installed from CRAN or some other repositories, the version number identifies it, and `packageURL = FALSE` will use the repository URL (as used by `utils::citation()`).

Value

A list containing the citations. Citations are also written to the `file` as a side effect.

Note

Some packages on CRAN do not have standard bib entries, which was once reported by Michael Friendly at https://stat.ethz.ch/pipermail/r-devel/2010-November/058977.html. I find this a pain, and there are no easy solutions except contacting package authors to modify their DESCRIPTION files. The argument `tweak` has provided hacks to deal with known packages with non-standard bib entries; `tweak = TRUE` is by no means intended to hide or modify the original citation information. It is just due to the loose requirements on package authors for the DESCRIPTION file. On one hand, I apologize if it really mangles the information about certain packages; on the other, I strongly recommend package authors to consider the `Authors@R` field (see the manual *Writing R Extensions*) to make it easier for other people to cite R packages.

Author(s)

Yihui Xie and Michael Friendly

Examples

```
pkg_bib(c("base", "MASS", "xfun"))
pkg_bib("cluster", prefix = "R-pkg-")  # a different prefix
pkg_bib("xfun", date = "package")
```

`process_file` *Read a text file, process the text with a function, and write the text back*

Description

Read a text file with the UTF-8 encoding, apply a function to the text, and write back to the original file if the processed text is different with the original input.

Usage

```
process_file(file, fun = identity, x = read_utf8(file))

sort_file(..., fun = sort)
```

Arguments

`file`	Path to a text file.
`fun`	A function to process the text.
`x`	The content of the file.
`...`	Arguments to be passed to `process_file()`.

Details

`sort_file()` is an application of `process_file()`, with the processing function being `sort()`, i.e., it sorts the text lines in a file and write back the sorted text.

Value

If `file` is provided, invisible `NULL` (the file is updated as a side effect), otherwise the processed content (as a character vector).

Examples

```
f = tempfile()
xfun::write_utf8("Hello World", f)
xfun::process_file(f, function(x) gsub("World", "woRld", x))
xfun::read_utf8(f)  # see if it has been updated
file.remove(f)
```

`proc_kill` *Kill a process and (optionally) all its child processes*

Description

Run the command `taskkill /f /pid` on Windows and `kill` on Unix, respectively, to kill a process.

Usage

```
proc_kill(pid, recursive = TRUE, ...)
```

Arguments

`pid`	The process ID.
`recursive`	Whether to kill the child processes of the process.
`...`	Arguments to be passed to `system2()` to run the command to kill the process.

Value

The status code returned from `system2()`.

`proj_root` *Return the (possible) root directory of a project*

Description

Given a path of a file (or dir) in a potential project (e.g., an R package or an RStudio project), return the path to the project root directory.

Usage

```
proj_root(path = "./", rules = root_rules)

root_rules
```

Arguments

`path`	The initial path to start the search. If it is a file path, its parent directory will be used.
`rules`	A matrix of character strings of two columns: the first column contains regular expressions to look for filenames that match the patterns, and the second column contains regular expressions to match the content of the matched files. The regular expression can be an empty string, meaning that it will match anything.

Format

An object of class `matrix` (inherits from `array`) with 2 rows and 2 columns.

Details

The search for the root directory is performed by a series of tests, currently including looking for a ‘`DESCRIPTION`’ file that contains `Package: *` (which usually indicates an R package), and a ‘`*.Rproj`’ file that contains `Version: *` (which usually indicates an RStudio project). If files with the expected patterns are not found in the initial directory, the search will be performed recursively in upper-level directories.

Value

Path to the root directory if found, otherwise `NULL`.

Note

This function was inspired by the **rprojroot** package, but is much less sophisticated. It is a rather simple function designed to be used in some of packages that I maintain, and may not meet the need of general users until this note is removed in the future (which should be unlikely). If you are sure that you are working on the types of projects mentioned in the ‘Details’ section, this function may be helpful to you, otherwise please consider using **rprojroot** instead.

`prose_index` *Find the indices of lines in Markdown that are prose (not code blocks)*

Description

Filter out the indices of lines between code block fences such as ```` ``` ```` (could be three or four or more backticks).

Usage

```
prose_index(x, warn = TRUE)
```

Arguments

`x`	A character vector of text in Markdown.
`warn`	Whether to emit a warning when code fences are not balanced.

Value

An integer vector of indices of lines that are prose in Markdown.

Note

If the code fences are not balanced (e.g., a starting fence without an ending fence), this function will treat all lines as prose.

Examples

```
library(xfun)
prose_index(c("a", "```", "b", "```", "c"))
prose_index(c("a", "````", "```r", "1+1", "```", "````", "c"))
```

---

protect_math *Protect math expressions in pairs of backticks in Markdown*

Description

For Markdown renderers that do not support LaTeX math, we need to protect math expressions as verbatim code (in a pair of backticks), because some characters in the math expressions may be interpreted as Markdown syntax (e.g., a pair of underscores may make text italic). This function detects math expressions in Markdown (by heuristics), and wrap them in backticks.

Usage

```
protect_math(x, token = "", use_block = FALSE)
```

Arguments

x	A character vector of text in Markdown.
token	A character string to wrap math expressions at both ends. This can be a unique token so that math expressions can be reliably identified and restored after the Markdown text is converted.
use_block	Whether to use code blocks (```md-math) to protect $$ $$ expressions that span across multiple lines. This is necessary when a certain line in the math expression starts with a special character that can accidentally start a new element (e.g., a leading + may start a bullet list). Only code blocks can prevent this case.

Details

Expressions in pairs of dollar signs or double dollar signs are treated as math, if there are no spaces after the starting dollar sign, or before the ending dollar sign. There should be a space or (before the starting dollar sign, unless the math expression starts from the very beginning of a line. For a pair of single dollar signs, the ending dollar sign should not be followed by a number, and the inner math expression should not be wrapped in backticks. With these assumptions, there should not be too many false positives when detecing math expressions.

Besides, LaTeX environments (\begin{*} and \end{*}) are also protected in backticks.

Value

A character vector with math expressions in backticks.

Note

If you are using Pandoc or the **rmarkdown** package, there is no need to use this function, because Pandoc's Markdown can recognize math expressions.

Examples

```
library(xfun)
protect_math(c("hi $a+b$", "hello $$\\alpha$$", "no math here: $x is $10 dollars"))
protect_math(c("hi $$", "\\begin{equation}", "x + y = z", "\\end{equation}"))
protect_math("$a+b$", "===")
```

`rand_unit` *Pseudo-random numbers on* `[0, 1)` *based on a linear congruential generator*

Description

Generate pseudo-random numbers from $X_{i+1} = (aX_i + c) \bmod m$, where X_1 is the initial value (seed).

Usage

```
rand_unit(n = 1, a = 55797, c = 0, m = 4294967296, seed = NULL)
```

Arguments

`n`	The desired length of the sequence.
`a, c, m`	Parameters for the generator (see References for the default values).
`seed`	The seed. The default is the system time when this function is called for the first time in the current session. For subsequent calls, the last X_i of the previous call will be used as the default seed.

Value

Random numbers on `[0, 1)` (i.e., X/m instead of X). Note the unit interval is open on the right (excluding 1).

Note

All argument values must be smaller than 2^{64} as they will be coerced to 64-bit integers.

References

Steele, Guy L. Jr.; Vigna, Sebastiano (2022). "Computationally easy, spectrally good multipliers for congruential pseudorandom number generators". *Software: Practice and Experience.* 52 (2): 443–458.

Examples

```
rand_unit(10)
rand_unit(10, seed = 0)
rand_unit(10, seed = 0)  # identical results
rand_unit(10, seed = Sys.getpid())
```

`raw_string` *Print a character vector in its raw form*

Description

The function `raw_string()` assigns the class `xfun_raw_string` to the character vector, and the corresponding printing function `print.xfun_raw_string()` uses `cat(x, sep = '\n')` to write the character vector to the console, which will suppress the leading indices (such as `[1]`) and double quotes, and it may be easier to read the characters in the raw form (especially when there are escape sequences).

Usage

```
raw_string(x, ...)

## S3 method for class 'xfun_raw_string'
print(x, ...)
```

Arguments

`x`	For `raw_string()`, a character vector. For the print method, the `raw_string()` object.
`...`	Other arguments (currently ignored).

Examples

```
library(xfun)
raw_string(head(LETTERS))
raw_string(c("a \"b\"", "hello\tworld!"))
```

read_all *Read all text files and concatenate their content*

Description

Read files one by one, and optionally add text before/after the content. Then combine all content into one character vector.

Usage

```
read_all(files, before = function(f, x) NULL, after = function(f, x) NULL)
```

Arguments

`files`	A vector of file paths.
`before, after`	A function that takes one file path and its content as the input and returns values to be added before or after the content of the file. Alternatively, they can be constant values to be added.

Value

A character vector.

Examples

```
# two files in this package
fs = system.file("scripts", c("call-fun.R", "child-pids.sh"), package = "xfun")
xfun::read_all(fs)

# add file paths before file content and an empty line after content
xfun::read_all(fs, before = function(f) paste("#-----", f, "-----"), after = "")

# add constants
xfun::read_all(fs, before = "/*", after = c("*/", ""))
```

read_bin *Read all records of a binary file as a raw vector by default*

Description

This is a wrapper function of `readBin()` with default arguments `what = "raw"` and `n = file.size(file)`, which means it will read the full content of a binary file as a raw vector by default.

Usage

```
read_bin(file, what = "raw", n = file.info(file)$size, ...)
```

Arguments

`file, what, n, ...`	Arguments to be passed to `readBin()`.

Value

A vector returned from `readBin()`.

Examples

```
f = tempfile()
cat("abc", file = f)
xfun::read_bin(f)
unlink(f)
```

`read_utf8` *Read / write files encoded in UTF-8*

Description

Read or write files, assuming they are encoded in UTF-8. `read_utf8()` is roughly `readLines(encoding = 'UTF-8')` (a warning will be issued if non-UTF8 lines are found), and `write_utf8()` calls `writeLines(enc2utf8(text), useBytes = TRUE)`.

Usage

```
read_utf8(con, error = FALSE)

write_utf8(text, con, ...)

append_utf8(text, con, sort = TRUE)

append_unique(text, con, sort = function(x) base::sort(unique(x)))
```

Arguments

`con`	A connection or a file path.
`error`	Whether to signal an error when non-UTF8 characters are detected (if `FALSE`, only a warning message is issued).
`text`	A character vector (will be converted to UTF-8 via `enc2utf8()`).
`...`	Other arguments passed to `writeLines()` (except `useBytes`, which is `TRUE` in `write_utf8()`).
`sort`	Logical (`FALSE` means not to sort the content) or a function to sort the content; `TRUE` is equivalent to `base::sort`.

Details

The function `append_utf8()` appends UTF-8 content to a file or connection based on `read_utf8()` and `write_utf8()`, and optionally sort the content. The function `append_unique()` appends unique lines to a file or connection.

Value

`read_utf8()` returns a character vector of the file content; `write_utf8()` returns the `con` argument (invisibly).

`record` *Run R code and record the results*

Description

Run R code and capture various types of output, including text output, plots, messages, warnings, and errors.

Usage

```
record(
  code = NULL,
  dev = "png",
  dev.path = "xfun-record",
  dev.ext = dev_ext(dev),
  dev.args = list(),
  dev.keep = TRUE,
  message = TRUE,
  warning = TRUE,
  error = NA,
  cache = list(),
  print = record_print,
  print.args = list(),
  verbose = getOption("xfun.record.verbose", 0),
  envir = parent.frame()
)

## S3 method for class 'xfun_record_results'
format(
  x,
  to = c("text", "markdown", "html"),
  encode = FALSE,
  template = FALSE,
  ...
)

## S3 method for class 'xfun_record_results'
```

```
print(
  x,
  browse = interactive(),
  to = if (browse) "html" else "text",
  template = TRUE,
  ...
)
```

Arguments

`code`	A character vector of R source code.
`dev`	A graphics device. It can be a function name, a function, or a character string that can be evaluated to a function to open a graphics device.
`dev.path`	A base file path for plots. Actual plot filenames will be this base path plus incremental suffixes. For example, if `dev.path = "foo"`, the plot files will be `foo-1.png`, `foo-2.png`, and so on. If `dev.path` is not character (e.g., `FALSE`), plots will not be recorded.
`dev.ext`	The file extension for plot files. By default, it will be inferred from the first argument of the device function if possible.
`dev.args`	Extra arguments to be passed to the device. The default arguments are `list(units = 'in', onefile = FALSE, width = 7, height = 7, res = 96)`. If any of these arguments is not present in the device function, it will be dropped.
`dev.keep`	Indices of plots to be kept. The indices can be either numeric (positive or negative integers) or logical. Negative integers and false values will remove the corresponding plots. For example, if the code generated 3 plots, `dev.keep = c(1, 3)`, `-2`, and `c(T, F, T)` are equivalent ways to remove the second plot. The special value `dev.keep = 'last'` means to keep the last plot only.
`message, warning, error`	If `TRUE`, record and store messages / warnings / errors in the output. If `FALSE`, suppress them. If `NA`, do not process them (messages will be emitted to the console, and errors will halt the execution).
`cache`	A list of options for caching. See the `path`, `id`, and `...` arguments of `cache_exec()`.
`print`	A (typically S3) function that takes the value of an expression in the code as input and returns output. The default is `record_print()`. If a non-function value (e.g., `NA`) is passed to this argument, `print()` (or `show()` for S4 objects) will be used.
`print.args`	A list of arguments for the `print` function. By default, the whole list is not passed directly to the function, but only an element in the list with a name identical to the first class name of the returned value of the expression, e.g., `list(data.frame = list(digits = 3), matrix = list())`. This makes it possible to apply different print arguments to objects of different classes. If the whole list is intended to be passed to the print function directly, wrap the list in `I()`.
`verbose`	`2` means to always print the value of each expression in the code, no matter if the value is `invisible()` or not; `1` means to always print the value of the last expression; `0` means no special handling (i.e., print only when the value is visible).

`envir`	An environment in which the code is evaluated.
`x`	An object returned by `record()`.
`to`	The output format (text, markdown, or html).
`encode`	For HTML output, whether to base64 encode plots.
`template`	For HTML output, whether to embed the formatted results in an HTML template. Alternatively, this argument can take a file path, i.e., path to an HTML template that contains the variable `$body$`. If `TRUE`, the default template in this package will be used (`xfun:::pkg_file('resources', 'record.html')`).
`...`	Currently ignored.
`browse`	Whether to browse the results on an HTML page.

Value

`record()` returns a list of the class `xfun_record_results` that contains elements with these possible classes: `record_source` (source code), `record_output` (text output), `record_plot` (plot file paths), `record_message` (messages), `record_warning` (warnings), and `record_error` (errors, only when the argument `error = TRUE`).

The `format()` method returns a character vector of plain-text output or HTML code for displaying the results.

The `print()` method prints the results as plain text or HTML to the console or displays the HTML page.

Examples

```
code = c("# a warning test", "1:2 + 1:3", "par(mar = c(4, 4, 1, .2))",
    "barplot(5:1, col = 2:6, horiz = TRUE)", "head(iris)",
    "sunflowerplot(iris[, 3:4], seg.col = 'purple')",
    "if (TRUE) {\n  message('Hello, xfun::record()!')\n}",
    "# throw an error", "1 + 'a'")
res = xfun::record(code, dev.args = list(width = 9, height = 6.75),
    error = TRUE)
xfun::tree(res)
format(res)
# find and clean up plot files
plots = Filter(function(x) inherits(x, "record_plot"),
    res)
file.remove(unlist(plots))
```

`record_print` *Print methods for* `record()`

Description

An S3 generic function to be called to print visible values in code when the code is recorded by `record()`. It is similar to `knitr::knit_print()`. By default, it captures the normal `print()` output and returns the result as a character vector. The `knitr_kable` method is for printing `knitr::kable()` output. Users and package authors can define other S3 methods to extend this function.

Usage

```
record_print(x, ...)

## Default S3 method:
record_print(x, ...)

## S3 method for class 'record_asis'
record_print(x, ...)

new_record(x, class)
```

Arguments

x	For `record_print()`, the value to be printed. For `new_record()`, a character vector to be included in the printed results.
...	Other arguments to be passed to `record_print()` methods.
class	A class name. Possible values are `xfun:::.record_cls`.

Value

A `record_print()` method should return a character vector or a list of character vectors. The original classes of the vector will be discarded, and the vector will be treated as console output by default (i.e., `new_record(class = "output")`). If it should be another type of output, wrap the vector in `new_record()` and specify a class name.

---

relative_path *Get the relative path of a path relative to a directory*

Description

Given a directory, return the relative path that is relative to this directory. For example, the path '`foo/bar.txt`' relative to the directory '`foo/`' is '`bar.txt`', and the path '`/a/b/c.txt`' relative to '`/d/e/`' is '`../../a/b/c.txt`'.

Usage

```
relative_path(x, dir = ".", use.. = TRUE, error = TRUE)
```

Arguments

x	A vector of paths to be converted to relative paths.
dir	Path to a directory.
use..	Whether to use double-dots ('`..`') in the relative path. A double-dot indicates the parent directory (starting from the directory provided by the `dir` argument).
error	Whether to signal an error if a path cannot be converted to a relative path.

Value

A vector of relative paths if the conversion succeeded; otherwise the original paths when `error = FALSE`, and an error when `error = TRUE`.

Examples

```
xfun::relative_path("foo/bar.txt", "foo/")
xfun::relative_path("foo/bar/a.txt", "foo/haha/")
xfun::relative_path(getwd())
```

`rename_seq` *Rename files with a sequential numeric prefix*

Description

Rename a series of files and add an incremental numeric prefix to the filenames. For example, files ‘`a.txt`’, ‘`b.txt`’, and ‘`c.txt`’ can be renamed to ‘`1-a.txt`’, ‘`2-b.txt`’, and ‘`3-c.txt`’.

Usage

```
rename_seq(
  pattern = "^[0-9]+-.+[.]Rmd$",
  format = "auto",
  replace = TRUE,
  start = 1,
  dry_run = TRUE
)
```

Arguments

`pattern`	A regular expression for `list.files()` to obtain the files to be renamed. For example, to rename `.jpeg` files, use `pattern = "[.]jpeg$"`.
`format`	The format for the numeric prefix. This is passed to `sprintf()`. The default format is `"\%0Nd"` where `N = floor(log10(n)) + 1` and `n` is the number of files, which means the prefix may be padded with zeros. For example, if there are 150 files to be renamed, the format will be `"\%03d"` and the prefixes will be `001`, `002`, ..., `150`.
`replace`	Whether to remove existing numeric prefixes in filenames.
`start`	The starting number for the prefix (it can start from 0).
`dry_run`	Whether to not really rename files. To be safe, the default is `TRUE`. If you have looked at the new filenames and are sure the new names are what you want, you may rerun `rename_seq()` with `dry_run = FALSE` to actually rename files.

Value

A named character vector. The names are original filenames, and the vector itself is the new filenames.

Examples

```
xfun::rename_seq()
xfun::rename_seq("[.](jpeg|png)$", format = "%04d")
```

---

rest_api *Get data from a REST API*

Description

Read data from a REST API and optionally with an authorization token in the request header. The function `rest_api_raw()` returns the raw text of the response, and `rest_api()` will parse the response with `jsonlite::fromJSON()` (assuming that the response is in the JSON format).

Usage

```
rest_api(...)

rest_api_raw(root, endpoint, token = "", params = list(), headers = NULL)

github_api(
  endpoint,
  token = "",
  params = list(),
  headers = NULL,
  raw = !loadable("jsonlite")
)
```

Arguments

`...`	Arguments to be passed to `rest_api_raw()`.
`root`	The API root URL.
`endpoint`	The API endpoint.
`token`	A named character string (e.g., `c(token = "xxxx")`), which will be used to create an authorization header of the form '`Authorization: NAME TOKEN`' for the API call, where '`NAME`' is the name of the string and '`TOKEN`' is the string. If the string does not have a name, '`Basic`' will be used as the default name.
`params`	A list of query parameters to be sent with the API call.
`headers`	A named character vector of HTTP headers, e.g., `c(Accept = "application/vnd.github.v3+json")`.
`raw`	Whether to return the raw response or parse the response with **jsonlite**.

Details

These functions are simple wrappers based on `url()` and `read_utf8()`. Specifically, the `headers` argument is passed to `url()`, and `read_utf8()` will send a 'GET' request to the API server. This means these functions only support the 'GET' method. If you need to use other HTTP methods (such as 'POST'), you have to use other packages such as **curl** and **httr**.

`github_api()` is a wrapper function based on `rest_api_raw()` to obtain data from the GitHub API: https://docs.github.com/en/rest. You can provide a personal access token (PAT) via the `token` argument, or via one of the environment variables *GITHUB_PAT*, *GITHUB_TOKEN*, *GH_TOKEN*. A PAT allows for a much higher rate limit in API calls. Without a token, you can only make 60 calls in an hour.

Value

A character vector (the raw JSON response) or an R object parsed from the JSON text.

Examples

```
# a normal GET request
xfun::rest_api("https://mockhttp.org", "/get")
xfun::rest_api_raw("https://mockhttp.org", "/get")

# send the request with an auth header
xfun::rest_api("https://mockhttp.org", "/headers", "OPEN SESAME!")

# with query parameters
xfun::rest_api("https://mockhttp.org", "/response-headers", params = list(foo = "bar"))

# get the rate limit info from GitHub
xfun::github_api("/rate_limit")
```

`retry` *Retry calling a function for a number of times*

Description

If the function returns an error, retry it for the specified number of times, with a pause between attempts.

Usage

```
retry(fun, ..., .times = 3, .pause = 5)
```

Arguments

`fun`	A function.
`...`	Arguments to be passed to the function.
`.times`	The number of times.
`.pause`	The number of seconds to wait before the next attempt.

Details

One application of this function is to download a web resource. Since the download might fail sometimes, you may want to retry it for a few more times.

Examples

```
# read the GitHub releases info of the repo yihui/xfun
xfun::retry(xfun::github_releases, "yihui/xfun")
```

---

rev_check *Run* R CMD check *on the reverse dependencies of a package*

Description

Install the source package, figure out the reverse dependencies on CRAN, download all of their source packages, and run R CMD check on them in parallel. The number of parallel downloads can be set via `options(xfun.rev_check.download_cores = n)` and the number of parallel checks can be set via `options(mc.cores = n)` (both default to `parallel::detectCores()`).

Usage

```
rev_check(
  pkg,
  which = "all",
  recheck = NULL,
  ignore = NULL,
  update = TRUE,
  timeout = getOption("xfun.rev_check.timeout", 15 * 60),
  src = file.path(src_dir, pkg),
  src_dir = getOption("xfun.rev_check.src_dir")
)

compare_Rcheck(status_only = TRUE, output = "00check_diffs.md")
```

Arguments

`pkg`	The package name.
`which`	Which types of reverse dependencies to check. See `tools::package_dependencies()` for possible values. The special value `'hard'` means the hard dependencies, i.e., `c('Depends', 'Imports', 'LinkingTo')`.
`recheck`	A vector of package names to be (re)checked. If not provided and there are any '`*.Rcheck`' directories left by certain packages (this often means these packages failed the last time), `recheck` will be these packages; if there are no '`*.Rcheck`' directories but a text file '`recheck`' exists, `recheck` will be the character vector read from this file. This provides a way for you to manually specify the packages to be checked. If there are no packages to be rechecked, all reverse dependencies will be checked.

`ignore`	A vector of package names to be ignored in `R CMD check`. If this argument is missing and a file '`00ignore`' exists, the file will be read as a character vector and passed to this argument.
`update`	Whether to update all packages before the check.
`timeout`	Timeout in seconds for `R CMD check` to check each package. The (approximate) total time can be limited by the global option `xfun.rev_check.timeout_total`.
`src`	The path of the source package directory.
`src_dir`	The parent directory of the source package directory. This can be set in a global option if all your source packages are under a common parent directory.
`status_only`	If `TRUE`, only compare the final statuses of the checks (the last line of '`00check.log`'), and delete '`*.Rcheck`' and '`*.Rcheck2`' if the statuses are identical, otherwise write out the full diffs of the logs. If `FALSE`, compare the full logs under '`*.Rcheck`' and '`*.Rcheck2`'.
`output`	The output Markdown file to which the diffs in check logs will be written. If the **markdown** package is available, the Markdown file will be converted to HTML, so you can see the diffs more clearly.

Details

Everything occurs under the current working directory, and you are recommended to call this function under a designated directory, especially when the number of reverse dependencies is large, because all source packages will be downloaded to this directory, and all '`*.Rcheck`' directories will be generated under this directory, too.

If a source tarball of the expected version has been downloaded before (under the '`tarball`' directory), it will not be downloaded again (to save time and bandwidth).

After a package has been checked, the associated '`*.Rcheck`' directory will be deleted if the check was successful (no warnings or errors or notes), which means if you see a '`*.Rcheck`' directory, it means the check failed, and you need to take a look at the log files under that directory.

The time to finish the check is recorded for each package. As the check goes on, the total remaining time will be roughly estimated via `n * mean(times)`, where `n` is the number of packages remaining to be checked, and `times` is a vector of elapsed time of packages that have been checked.

If a check on a reverse dependency failed, its '`*.Rcheck`' directory will be renamed to '`*.Rcheck2`', and another check will be run against the CRAN version of the package unless `options(xfun.rev_check.compare = FALSE)` is set. If the logs of the two checks are the same, it means no new problems were introduced in the package, and you can probably ignore this particular reverse dependency. The function `compare_Rcheck()` can be used to create a summary of all the differences in the check logs under '`*.Rcheck`' and '`*.Rcheck2`'. This will be done automatically if `options(xfun.rev_check.summary = TRUE)` has been set.

A recommended workflow is to use a special directory to run `rev_check()`, set the global `options()` `xfun.rev_check.src_dir` and `repos` in the R startup (see `?Startup`) profile file `.Rprofile` under this directory, and (optionally) set `R_LIBS_USER` in '`.Renviron`' to use a special library path (so that your usual library will not be cluttered). Then run `xfun::rev_check(pkg)` once, investigate and fix the problems or (if you believe it was not your fault) ignore broken packages in the file '`00ignore`', and run `xfun::rev_check(pkg)` again to recheck the failed packages. Repeat this process until all '`*.Rcheck`' directories are gone.

As an example, I set `options(repos = c(CRAN = 'https://cran.rstudio.com'), xfun.rev_check.src_dir = '~/Dropbox/repo')` in '`.Rprofile`', and `R_LIBS_USER=~/R-tmp` in '`.Renviron`'. Then I can run, for example, `xfun::rev_check('knitr')` repeatedly under a special directory '`~/Downloads/revcheck`'. Reverse dependencies and their dependencies will be installed to '`~/R-tmp`', and **knitr** will be installed from '`~/Dropbox/repo/kintr`'.

Value

A named numeric vector with the names being package names of reverse dependencies; `0` indicates check success, `1` indicates failure, and `2` indicates that a package was not checked due to global timeout.

See Also

`devtools::revdep_check()` is more sophisticated, but currently has a few major issues that affect me: (1) It always deletes the '`*.Rcheck`' directories (`https://github.com/r-lib/devtools/issues/1395`), which makes it difficult to know more information about the failures; (2) It does not fully install the source package before checking its reverse dependencies (`https://github.com/r-lib/devtools/pull/1397`); (3) I feel it is fairly difficult to iterate the check (ignore the successful packages and only check the failed packages); by comparison, `xfun::rev_check()` only requires you to run a short command repeatedly (failed packages are indicated by the existing '`*.Rcheck`' directories, and automatically checked again the next time).

`xfun::rev_check()` borrowed a very nice feature from `devtools::revdep_check()`: estimating and displaying the remaining time. This is particularly useful for packages with huge numbers of reverse dependencies.

`Rscript` *Run the commands* `Rscript` *and* `R CMD`

Description

Wrapper functions to run the commands `Rscript` and `R CMD`.

Usage

```
Rscript(args, ...)

Rcmd(args, ...)
```

Arguments

`args`	A character vector of command-line arguments.
`...`	Other arguments to be passed to `system2()`.

Value

A value returned by `system2()`.

Examples

```
library(xfun)
Rscript(c("-e", "1+1"))
Rcmd(c("build", "--help"))
```

`Rscript_call` *Call a function in a new R session via* `Rscript()`

Description

Save the argument values of a function in a temporary RDS file, open a new R session via `Rscript()`, read the argument values, call the function, and read the returned value back to the current R session.

Usage

```
Rscript_call(
  fun,
  args = list(),
  options = NULL,
  ...,
  wait = TRUE,
 fail = sprintf("Failed to run '%s' in a new R session", deparse(substitute(fun))[1])
)
```

Arguments

`fun`	A function, or a character string that can be parsed and evaluated to a function.
`args`	A list of argument values.
`options`	A character vector of options to passed to `Rscript()`, e.g., `"--vanilla"`.
`..., wait`	Arguments to be passed to `system2()`.
`fail`	The desired error message when an error occurred in calling the function. If the actual error message during running the function is available, it will be appended to this message.

Value

If `wait = TRUE`, the returned value of the function in the new R session. If `wait = FALSE`, three file paths will be returned: the first one stores `fun` and `args` (as a list), the second one is supposed to store the returned value of the function, and the third one stores the possible error message.

Examples

```
factorial(10)
# should return the same value
xfun::Rscript_call("factorial", list(10))

# the first argument can be either a character string or a function
xfun::Rscript_call(factorial, list(10))

# Run Rscript starting a vanilla R session
xfun::Rscript_call(factorial, list(10), options = c("--vanilla"))
```

`rstudio_type` *Type a character vector into the RStudio source editor*

Description

Use the **rstudioapi** package to insert characters one by one into the RStudio source editor, as if they were typed by a human.

Usage

```
rstudio_type(x, pause = function() 0.1, mistake = 0, save = 0)
```

Arguments

x	A character vector.
pause	A function to return a number in seconds to pause after typing each character.
mistake	The probability of making random mistakes when typing the next character. A random mistake is a random string typed into the editor and deleted immediately.
save	The probability of saving the document after typing each character. Note that If a document is not opened from a file, it will never be saved.

Examples

```
library(xfun)
if (loadable("rstudioapi") && rstudioapi::isAvailable()) {
    rstudio_type("Hello, RStudio! xfun::rstudio_type() looks pretty cool!",
        pause = function() runif(1, 0, 0.5), mistake = 0.1)
}
```

`same_path` *Test if two paths are the same after they are normalized*

Description

Compare two paths after normalizing them with the same separator (`/`).

Usage

```
same_path(p1, p2, ...)
```

Arguments

`p1, p2`	Two vectors of paths.
`...`	Arguments to be passed to `normalize_path()`.

Examples

```
library(xfun)
same_path("~/foo", file.path(Sys.getenv("HOME"), "foo"))
```

`session_info` *An alternative to sessionInfo() to print session information*

Description

This function tweaks the output of `sessionInfo()`: (1) It adds the RStudio version information if running in the RStudio IDE; (2) It removes the information about matrix products, BLAS, and LAPACK; (3) It removes the names of base R packages; (4) It prints out package versions in a single group, and does not differentiate between loaded and attached packages.

Usage

```
session_info(packages = NULL, dependencies = TRUE)
```

Arguments

`packages`	A character vector of package names, of which the versions will be printed. If not specified, it means all loaded and attached packages in the current R session.
`dependencies`	Whether to print out the versions of the recursive dependencies of packages.

Details

It also allows you to only print out the versions of specified packages (via the `packages` argument) and optionally their recursive dependencies. For these specified packages (if provided), if a function `xfun_session_info()` exists in a package, it will be called and expected to return a character vector to be appended to the output of `session_info()`. This provides a mechanism for other packages to inject more information into the `session_info` output. For example, **rmarkdown** (>= 1.20.2) has a function `xfun_session_info()` that returns the version of Pandoc, which can be very useful information for diagnostics.

Value

A character vector of the session information marked as `raw_string()`.

Examples

```
xfun::session_info()
if (xfun::loadable("MASS")) xfun::session_info("MASS")
```

set_envvar *Set environment variables*

Description

Set environment variables from a named character vector, and return the old values of the variables, so they could be restored later.

Usage

```
set_envvar(vars)
```

Arguments

`vars`	A named character vector of the form `c(VARIABLE = VALUE)`. If any value is `NA`, this function will try to unset the variable.

Details

The motivation of this function is that `Sys.setenv()` does not return the old values of the environment variables, so it is not straightforward to restore the variables later.

Value

Old values of the variables (if not set, `NA`).

Examples

```
vars = xfun::set_envvar(c(FOO = "1234"))
Sys.getenv("FOO")
xfun::set_envvar(vars)
Sys.getenv("FOO")
```

---

shrink_images *Shrink images to a maximum width*

Description

Use `magick::image_resize()` to shrink an image if its width is larger than the value specified by the argument `width`, and optionally call `tinify()` to compress it.

Usage

```
shrink_images(
  width = 800,
  dir = ".",
  files = all_files("[.](png|jpe?g|webp)$", dir),
  tinify = FALSE
)
```

Arguments

`width`	The desired maximum width of images.
`dir`	The directory of images.
`files`	A vector of image file paths. By default, this is all '`.png`', '`.jpeg`', and '`.webp`' images under `dir`.
`tinify`	Whether to compress images using `tinify()`.

Examples

```
f = xfun:::all_files("[.](png|jpe?g)$", R.home("doc"))
file.copy(f, tempdir())
f = file.path(tempdir(), basename(f))
magick::image_info(magick::image_read(f))  # some widths are larger than 300
xfun::shrink_images(300, files = f)
magick::image_info(magick::image_read(f))  # all widths <= 300 now
file.remove(f)
```

split_lines *Split a character vector by line breaks*

Description

Call `unlist(strsplit(x, '\n'))` on the character vector `x` and make sure it works in a few edge cases: `split_lines('')` returns `''` instead of `character(0)` (which is the returned value of `strsplit('', '\n')`); `split_lines('a\n')` returns `c('a', '')` instead of `c('a')` (which is the returned value of `strsplit('a\n', '\n')`.

Usage

```
split_lines(x)
```

Arguments

`x`	A character vector.

Value

All elements of the character vector are split by `'\n'` into lines.

Examples

```
xfun::split_lines(c("a", "b\nc"))
```

split_source *Split source lines into complete expressions*

Description

Parse the lines of code one by one to find complete expressions in the code, and put them in a list.

Usage

```
split_source(x, merge_comments = FALSE, line_number = FALSE)
```

Arguments

`x`	A character vector of R source code.
`merge_comments`	Whether to merge consecutive lines of comments as a single expression to be combined with the next non-comment expression (if any).
`line_number`	Whether to store the line numbers of each expression in the returned value.

Value

A list of character vectors, and each vector contains a complete R expression, with an attribute `lines` indicating the starting and ending line numbers of the expression if the argument `line_number = TRUE`.

Examples

```
code = c("# comment 1", "# comment 2", "if (TRUE) {", "1 + 1", "}", "print(1:5)")
xfun::split_source(code)
xfun::split_source(code, merge_comments = TRUE)
```

`strict_list` *Strict lists*

Description

A strict list is essentially a normal `list()` but it does not allow partial matching with `$`.

Usage

```
strict_list(...)

as_strict_list(x)

## S3 method for class 'xfun_strict_list'
x$name

## S3 method for class 'xfun_strict_list'
print(x, ...)
```

Arguments

`...`	Objects (list elements), possibly named. Ignored in the `print()` method.
`x`	For `as_strict_list()`, the object to be coerced to a strict list. For `print()`, a strict list.
`name`	The name (a character string) of the list element.

Details

To me, partial matching is often more annoying and surprising than convenient. It can lead to bugs that are very hard to discover, and I have been bitten by it many times. When I write `x$name`, I always mean precisely `name`. You should use a modern code editor to autocomplete the `name` if it is too long to type, instead of using partial names.

Value

Both `strict_list()` and `as_strict_list()` return a list with the class `xfun_strict_list`. Whereas `as_strict_list()` attempts to coerce its argument `x` to a list if necessary, `strict_list()` just wraps its argument `...` in a list, i.e., it will add another list level regardless if `...` already is of type list.

Examples

```
library(xfun)
(z = strict_list(aaa = "I am aaa", b = 1:5))
z$a  # NULL!
z$aaa  # I am aaa
z$b
z$c = "create a new element"

z2 = unclass(z)  # a normal list
z2$a  # partial matching

z3 = as_strict_list(z2)  # a strict list again
z3$a  # NULL again!
```

`strip_html` *Strip HTML tags*

Description

Remove HTML tags and comments from text.

Usage

```
strip_html(x)
```

Arguments

`x`	A character vector.

Value

A character vector with HTML tags and comments stripped off.

Examples

```
xfun::strip_html("<a href=\"#\">Hello <!-- comment -->world!</a>")
```

`str_wrap` *Wrap character vectors*

Description

A wrapper function to make `strwrap()` return a character vector of the same length as the input vector; each element of the output vector is a string formed by concatenating wrapped strings by `\n`.

Usage

```
str_wrap(...)
```

Arguments

`...`	Arguments passed to `strwrap()`.

Value

A character vector.

Examples

```
x = sample(c(letters, " "), 200, TRUE, c(rep(0.5/26, 26), 0.5))
x = rep(paste(x, collapse = ""), 2)
strwrap(x, width = 30)
xfun::str_wrap(x, width = 30)  # same length as x
```

`submit_cran` *Submit a source package to CRAN*

Description

Build a source package and submit it to CRAN with the **curl** package.

Usage

```
submit_cran(file = pkg_build(), comment = "")
```

Arguments

`file`	The path to the source package tarball. By default, the current working directory is treated as the package root directory, and automatically built into a tarball, which is deleted after submission. This means you should run `xfun::submit_cran()` in the root directory of a package project, unless you want to pass a path explicitly to the `file` argument.
`comment`	Submission comments for CRAN. By default, if a file ‘`cran-comments.md`’ exists, its content will be read and used as the comment.

See Also

`devtools::submit_cran()` does the same job, with a few more dependencies in addition to **curl** (such as **cli**); `xfun::submit_cran()` only depends on **curl**.

`system3` *Run `system2()` and mark its character output as UTF-8 if appropriate*

Description

This is a wrapper function based on `system2()`. If `system2()` returns character output (e.g., with the argument `stdout = TRUE`), check if the output is encoded in UTF-8. If it is, mark it with UTF-8 explicitly.

Usage

```
system3(...)
```

Arguments

`...`	Passed to `system2()`.

Value

The value returned by `system2()`.

Examples

```
a = shQuote(c("-e", "print(intToUtf8(c(20320, 22909)))"))
x2 = system2("Rscript", a, stdout = TRUE)
Encoding(x2)  # unknown

x3 = xfun::system3("Rscript", a, stdout = TRUE)
# encoding of x3 should be UTF-8 if the current locale is UTF-8
!l10n_info()[["UTF-8"]] || Encoding(x3) == "UTF-8"  # should be TRUE
```

`tabset` *Represent a (recursive) list with (nested) tabsets*

Description

The tab titles are names of list members, and the tab content contains the values of list members. If a list member is also a list, it will be represented recursively with a child tabset.

Usage

```
tabset(x, value = str)
```

Arguments

`x`	A list.
`value`	A function to print the value of a list member. By default, `str()` is used to print the structure of the value. You may also use `dput()` to output the full value, but it may be slow when the size of the value is too big.

Value

A character vector of Markdown that can be rendered to HTML with `litedown::mark()`.

Examples

```
xfun::tabset(iris)
xfun::tabset(iris, dput)
xfun::tabset(iris, print)

# a deeply nested list
plot(1:10)
p = recordPlot()
xfun::tabset(p)
```

`taml_load` *A simple YAML reader and writer*

Description

TAML is a tiny subset of YAML. See `https://yihui.org/litedown/#sec:yaml-syntax` for its specifications.

Usage

```
taml_load(x, envir = parent.frame())

taml_file(path)

taml_save(x, path = NULL, indent = "  ")
```

Arguments

`x`	For `taml_load()`, a character vector of the YAML content. For `taml_save()`, a list to be converted to YAML.
`envir`	The environment in which R expressions in YAML are evaluated. To disable the evaluation, use `envir = FALSE`.
`path`	A file path to read from or write to.
`indent`	A character string to indent sub-lists by one level.

Value

`taml_load()` and `taml_file()` return a list; if `path = NULL`, `taml_save()` returns a character vector, otherwise the vector is written to the file specified by the `path`.

Examples

```
(res = taml_load("a: 1"))
taml_save(res)

(res = taml_load("a: 1\nb: \"foo\"\nc: null"))
taml_save(res)

(res = taml_load("a:\n  b: false\n  c: true\n  d: 1.234\ne: bar"))
taml_save(res)
taml_save(res, indent = "\t")

taml_load("a: !expr paste(1:10, collapse = \", \")")
taml_load("a: [1, 3, 4, 2]")
taml_load("a: [1, \"abc\", 4, 2]")
taml_load("a: [\"foo\", \"bar\"]")
taml_load("a: [true, false, true]")
# the other form of array is not supported
taml_load("a:\n  - b\n  - c")
# and you must use the yaml package
if (loadable("yaml")) yaml_load("a:\n  - b\n  - c")
```

---

tinify *Use the Tinify API to compress PNG and JPEG images*

Description

Compress PNG/JPEG images with ‘`api.tinify.com`’, and download the compressed images. These functions require R packages **curl** and **jsonlite**. `tinify_dir()` is a wrapper function of `tinify()` to compress images under a directory.

Usage

```
tinify(
  input,
  output,
  quiet = FALSE,
  force = FALSE,
  key = env_option("xfun.tinify.key"),
  history = env_option("xfun.tinify.history")
)

tinify_dir(dir = ".", ...)
```

Arguments

`input`	A vector of input paths of images.
`output`	A vector of output paths or a function that takes `input` and returns a vector of output paths (e.g., `output = identity` means `output = input`). By default, if the `history` argument is not a provided, `output` is `input` with a suffix `-min` (e.g., when `input = 'foo.png'`, `output = 'foo-min.png'`), otherwise `output` is the same as `input`, which means the original image files will be overwritten.
`quiet`	Whether to suppress detailed information about the compression, which is of the form '`input.png (10 Kb) ==> output.png (5 Kb, 50%); compression count: 42`'. The percentage after `output.png` stands for the compression ratio, and the compression count shows the number of compressions used for the current month.
`force`	Whether to compress an image again when it appears to have been compressed before. This argument only makes sense when the `history` argument is provided.
`key`	The Tinify API key. It can be set via either the global option `xfun.tinify.key` or the environment variable `R_XFUN_TINIFY_KEY` (see `env_option()`).
`history`	Path to a history file to record the MD5 checksum of compressed images. If the checksum of an expected output image exists in this file and `force = FALSE`, the compression will be skipped. This can help you avoid unnecessary API calls.
`dir`	A directory under which all '`.png`', '`.jpeg`', and '`.webp`' files are to be compressed.
`...`	Arguments passed to `tinify()`.

Details

You are recommended to set the API key in '`.Rprofile`' or '`.Renviron`'. After that, the only required argument of this function is `input`. If the original images can be overwritten by the compressed images, you may either use `output = identity`, or set the value of the `history` argument in '`.Rprofile`' or '`.Renviron`'.

Value

The output file paths.

References

Tinify API: `https://tinypng.com/developers`.

See Also

The **tinieR** package (`https://github.com/jmablog/tinieR/`) is a more comprehensive implementation of the Tinify API, whereas `xfun::tinify()` has only implemented the feature of shrinking images.

Examples

```
## Not run:
f = xfun::R_logo("jpg$")
xfun::tinify(f)  # remember to set the API key before trying this

## End(Not run)
```

`tojson` *A simple JSON serializer*

Description

A JSON serializer that only works on a limited types of R data (`NULL`, lists, arrays, logical/character/numeric/date/time vectors). Other types of data will be coerced to character. A character string of the class `JS_LITERAL` is treated as raw JavaScript, so will not be quoted. The function `json_vector()` converts an atomic R vector to JSON.

Usage

```
tojson(x)

json_vector(x, to_array = FALSE, quote = TRUE)
```

Arguments

`x`	An R object.
`to_array`	Whether to convert a vector to a JSON array (use `[]`).
`quote`	Whether to double quote the elements.

Details

Both `NULL` and `NA` are converted to `null`. Named lists are converted to objects of the form `{key1: value1, key2: value2, .`
Unnamed lists are converted to arrays of the form `[[value1], [value2], ...]`. The same rules apply to data frames since technically they are also lists. However, please note that unnamed data frames (i.e., without column names) will be converted to an array with each *row* as an array element, whereas named data frames will have each *column* as an individual element. For matrices, the JSON array will have each row as an individual element, and names are discarded.

Dates and times are coerced to character using UTC as the timezone, and represented via the JavaScript expression `new Date(value)` (which is not standard JSON but practically more useful).

Value

A character string.

See Also

The **jsonlite** package provides a full JSON serializer.

Examples

```
library(xfun)
tojson(NULL)
tojson(1:10)
tojson(TRUE)
tojson(FALSE)
tojson(list(a = 1, b = list(c = 1:3, d = "abc")))
tojson(list(c("a", "b"), 1:5, TRUE, Sys.Date() + 1:3))
tojson(head(iris))  # each column is in an element
tojson(unname(head(iris)))  # each row is in an element
tojson(matrix(1:12, 3))

# literal JS code
JS = function(x) structure(x, class = "JS_LITERAL")
tojson(list(a = 1:5, b = JS("function() {return true;}")))
```

`tree` *Turn the output of `str()` into a tree diagram*

Description

The super useful function `str()` uses `..` to indicate the level of sub-elements of an object, which may be difficult to read. This function uses vertical pipes to connect all sub-elements on the same level, so it is clearer which elements belong to the same parent element in an object with a nested structure (such as a nested list).

Usage

```
tree(...)
```

Arguments

`...`	Arguments to be passed to `str()` (note that the `comp.str` is hardcoded inside this function, and it is the only argument that you cannot customize).

Value

A character string as a `raw_string()`.

Examples

```
fit = lsfit(1:9, 1:9)
str(fit)
xfun::tree(fit)

fit = lm(dist ~ speed, data = cars)
str(fit)
xfun::tree(fit)

# some trivial examples
xfun::tree(1:10)
xfun::tree(iris)
```

`try_error` *Try an expression and see if it throws an error*

Description

Use `tryCatch()` to check if an expression throws an error.

Usage

```
try_error(expr)
```

Arguments

`expr`	An R expression.

Value

`TRUE` (error) or `FALSE` (success).

Examples

```
xfun::try_error(stop("foo"))  # TRUE
xfun::try_error(1:10)  # FALSE
```

`try_silent` *Try to evaluate an expression silently*

Description

An abbreviation of `try(silent = TRUE)`.

Usage

```
try_silent(expr)
```

Arguments

`expr`	An R expression.

Examples

```
library(xfun)
z = try_silent(stop("Wrong!"))
inherits(z, "try-error")
```

`upload_ftp` *Upload to an FTP server via* `curl`

Description

The function `upload_ftp()` runs the command `curl -T file server` to upload a file to an FTP server if the system command `curl` is available, otherwise it uses the R package **curl**. The function `upload_win_builder()` uses `upload_ftp()` to upload packages to the win-builder server.

Usage

```
upload_ftp(file, server, dir = "")

upload_win_builder(
  file = pkg_build(),
  version = c("R-devel", "R-release", "R-oldrelease"),
  server = c("ftp", "https")
)
```

Arguments

`file`	Path to a local file.
`server`	The address of the FTP server. For `upload_win_builder()`, `server = 'https'` means uploading to `'https://win-builder.r-project.org/upload.aspx'`.
`dir`	The remote directory to which the file should be uploaded.
`version`	The R version(s) on win-builder.

Details

These functions were written mainly to save package developers the trouble of going to the win-builder web page and uploading packages there manually.

Value

Status code returned from `system2()` or `curl::curl_fetch_memory()`.

`upload_imgur` *Upload an image to imgur.com*

Description

This function uses the **curl** package or the system command `curl` (whichever is available) to upload a image to `https://imgur.com`.

Usage

```
upload_imgur(
  file,
  key = env_option("xfun.upload_imgur.key"),
  use_curl = loadable("curl"),
  include_xml = FALSE
)
```

Arguments

`file`	Path to the image file to be uploaded.
`key`	Client ID for Imgur. It can be set via either the global option `xfun.upload_imgur.key` or the environment variable `R_XFUN_UPLOAD_IMGUR_KEY` (see `env_option()`). If neither is set, this uses a client ID registered by Yihui Xie.
`use_curl`	Whether to use the R package **curl** to upload the image. If `FALSE`, the system command `curl` will be used.
`include_xml`	Whether to include the XML response in the returned value.

Details

One application is to upload local image files to Imgur when knitting a document with **knitr**: you can set the `knitr::opts_knit$set(upload.fun = xfun::upload_imgur`, so the output document does not need local image files any more, and it is ready to be published online.

Value

A character string of the link to the image. If `include_xml = TRUE`, this string carries an attribute named `XML`, which is the XML response from Imgur (it will be parsed by **xml2** if available). See Imgur API in the references.

Note

Please register your own Imgur application to get your client ID. You can certainly use mine, but this ID is in the public domain and it might reach Imgur's rate limit if too many people are using it in the same time period or someone is trying to upload too many images at once.

Author(s)

Yihui Xie, adapted from the **imguR** package by Aaron Statham

References

A demo: https://yihui.org/knitr/demo/upload/

Examples

```
## Not run:
f = tempfile(fileext = ".png")
png(f)
plot(rnorm(100), main = R.version.string)
dev.off()

res = imgur_upload(f, include_xml = TRUE)
res  # link to original URL of the image
xfun::attr2(res, "XML")  # all information
if (interactive())
    browseURL(res)

# to use your own key
options(xfun.upload_imgur.key = "your imgur key")

## End(Not run)
```

url_accessible *Test if a URL is accessible*

Description

Try to send a HEAD request to a URL using curlGetHeaders() or the **curl** package, and see if it returns a successful status code.

Usage

```
url_accessible(x, use_curl = !capabilities("libcurl"), ...)
```

Arguments

`x`	A URL as a character string.
`use_curl`	Whether to use the **curl** package or the `curlGetHeaders()` function in base R to send the request to the URL. By default, **curl** will be used when base R does not have the `libcurl` capability (which should be rare).
`...`	Arguments to be passed to `curlGetHeaders()`.

Value

`TRUE` or `FALSE`.

Examples

```
xfun::url_accessible("https://yihui.org")
```

`url_destination` *Get the final destination of a URL*

Description

If a URL is redirected, query the new location via `curlGetHeaders()`, otherwise return the original URL.

Usage

```
url_destination(x, force = FALSE)
```

Arguments

`x`	A character vector of URLs.
`force`	By default, the query is cached in the current R session. To bypass the cache, use `force = TRUE`.

Value

The final destination(s). If the URL returns a status code greater than or equal to 400, it will throw an error.

Examples

```
u = "https://tinytex.yihui.org"  # redirected to https://yihui.org/tinytex/
if (url_accessible(u)) url_destination(u)
```

`url_filename` *Extract filenames from a URLs*

Description

Get the base names of URLs via `basename()`, and remove the possible query parameters or hash from the names.

Usage

```
url_filename(x, default = "index.html")
```

Arguments

`x`	A character vector of URLs.
`default`	The default filename when it cannot be determined from the URL, e.g., when the URL ends with a slash.

Value

A character vector of filenames at the end of URLs.

Examples

```
xfun::url_filename("https://yihui.org/images/logo.png")
xfun::url_filename("https://yihui.org/index.html")
xfun::url_filename("https://yihui.org/index.html?foo=bar")
xfun::url_filename("https://yihui.org/index.html#about")
xfun::url_filename("https://yihui.org")
xfun::url_filename("https://yihui.org/")
```

`valid_syntax` *Check if the syntax of the code is valid*

Description

Try to `parse()` the code and see if an error occurs.

Usage

```
valid_syntax(code, silent = TRUE)
```

Arguments

`code`	A character vector of R source code.
`silent`	Whether to suppress the error message when the code is not valid.

Value

TRUE if the code could be parsed, otherwise FALSE.

Examples

```
xfun::valid_syntax("1+1")
xfun::valid_syntax("1+")
xfun::valid_syntax(c("if(T){1+1}", "else {2+2}"), silent = FALSE)
```

---

yaml_body *Partition the YAML metadata and the body in a document*

Description

Split a document into the YAML metadata (which starts with `---` in the beginning of the document) and the body.

Usage

```
yaml_body(x, ..., parse = TRUE)
```

Arguments

x	A character vector of the document content.
...	Arguments to be passed to `yaml_load()`.
parse	Whether to parse the YAML data.

Value

A list of components `yaml` (the YAML data), `lines` (starting and ending line numbers of YAML), and `body` (a character vector of the body text). If YAML metadata does not exist in the document, the components `yaml` and `lines` will be missing.

Examples

```
xfun::yaml_body(c("---", "title: Hello", "output: litedown::html_format", "---",
    "", "Content."))
```

yaml_load — *Read YAML data*

Description

If the **yaml** package is installed, use `yaml::yaml.load()` to read the data. If not, use the simple parser `taml_load()` instead.

Usage

```
yaml_load(
  x,
  ...,
  handlers = NULL,
  envir = parent.frame(),
  use_yaml = loadable("yaml")
)
```

Arguments

`x`	A character vector of YAML data.
`...`, `handlers`	Arguments to be passed to `yaml::yaml.load()`.
`envir`	The environment in which R expressions in YAML are evaluated. To disable the evaluation, use `envir = FALSE`.
`use_yaml`	Whether to use the **yaml** package.

Value

An R object (typically a list).

Note

R expressions in YAML will be returned as expressions when they are not evaluated. This is different with `yaml::yaml.load()`, which returns character strings for expressions.

Examples

```
yaml_load("a: 1")
yaml_load("a: 1", use_yaml = FALSE)
```

Index