Package 'xfun'

January 6, 2021

Type Package

Title Miscellaneous Functions by 'Yihui Xie'

Version 0.20

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

Imports stats, tools

Suggests testit, parallel, codetools, rstudioapi, tinytex, mime, markdown, knitr, htmltools, remotes, pak, rmarkdown

License MIT + file LICENSE

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

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

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

VignetteBuilder knitr

NeedsCompilation yes

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

Maintainer Yihui Xie <xie@yihui.name>

Repository CRAN

Date/Publication 2021-01-06 16:00:03 UTC

R topics documented:

`attr` *Obtain an attribute of an object without partial matching*

Description

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

Usage

```
attr(...)
```

Arguments

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

Examples

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

Arguments

x	A file path.

Value

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

Examples

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

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())
```

Arguments

`command, args`	The system command and its arguments. They do not need to be quoted, since they will be quoted via `shQuote()` internally.

Details

If you need to see the output from stdout and stderr, you may set `options(xfun.bg_process.verbose = TRUE)` before starting the process. By default, these outputs are hidden.

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.

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_rds *Cache the value of an R expression to an RDS file*

Description

Save the value of an expression to a cache file (of the RDS format). Next time the value is loaded from the file if it exists.

Usage

```
cache_rds(
  expr = { },
  rerun = FALSE,
  file = "cache.rds",
  dir = "cache/",
  hash = NULL,
  clean = getOption("xfun.cache_rds.clean", TRUE),
  ...
)
```

Arguments

expr	An R expression.
rerun	Whether to delete the RDS file, rerun the expression, and save the result again (i.e., invalidate the cache if it exists).
file	The *base* (see Details) cache filename under the directory specified by the dir argument. If not specified and this function is called inside a code chunk of a **knitr** document (e.g., an R Markdown document), the default is the current chunk label plus the extension '.rds'.
dir	The path of the RDS file is partially determined by paste0(dir,file). If not specified and the **knitr** package is available, the default value of dir is the **knitr** chunk option cache.path (so if you are compiling a **knitr** document, you do not need to provide this dir argument explicitly), otherwise the default is 'cache/'. If you do not want to provide a dir but simply a valid path to the file argument, you may use dir = "".
hash	A list object that contributes to the MD5 hash of the cache filename (see Details). It can also take a special character value "auto". Other types of objects are ignored.
clean	Whether to clean up the old cache files automatically when expr has changed.
...	Other arguments to be passed to saveRDS().

Details

Note that the file argument does not provide the full cache filename. The actual name of the cache file is of the form 'BASENAME_HASH.rds', where 'BASENAME' is the base name provided via the 'file' argument (e.g., if file = 'foo.rds', BASENAME would be 'foo'), and 'HASH' is the MD5 hash (also called the 'checksum') calculated from the R code provided to the expr argument and the value of the hash argument, which means when the code or the hash argument changes, the 'HASH' string may also change, and the old cache will be invalidated (if it exists). If you want to find the cache file, look for '.rds' files that contain 32 hexadecimal digits (consisting of 0-9 and a-z) at the end of the filename.

The possible ways to invalidate the cache are: 1) change the code in expr argument; 2) delete the cache file manually or automatically through the argument rerun = TRUE; and 3) change the value of the hash argument. The first two ways should be obvious. For the third way, it makes it possible to automatically invalidate the cache based on changes in certain R objects. For example, when you run cache_rds({ x + y }), you may want to invalidate the cache to rerun { x + y } when the value of x or y has been changed, and you can tell cache_rds() to do so by cache_rds({ x + y },hash = list(x,y)). The value of the argument hash is expected to be a list, but it can also take a special value, "auto", which means cache_rds(expr) will try to automatically figure out the global variables in expr, return a list of their values, and use this list as the actual value of hash. This behavior is most likely to be what you really want: if the code in expr uses an external global variable, you may want to invalidate the cache if the value of the global variable has changed. Here a "global variable" means a variable not created locally in expr, e.g., for cache_rds({ x <-1; x + y }), x is a local variable, and y is (most likely to be) a global variable, so changes in y should invalidate the cache. However, you know your own code the best. If you want to be completely sure when to invalidate the cache, you can always provide a list of objects explicitly rather than relying on hash = "auto".

By default (the argument `clean = TRUE`), old cache files will be automatically cleaned up. Sometimes you may want to use `clean = FALSE` (set the R global option `options(xfun.cache_rds.clean = FALSE)` if you want `FALSE` to be the default). For example, you may not have decided which version of code to use, and you can keep the cache of both versions with `clean = FALSE`, so when you switch between the two versions of code, it will still be fast to run the code.

Value

If the cache file does not exist, run the expression and save the result to the file, otherwise read the cache file and return the value.

Note

Changes in the code in the `expr` argument do not necessarily always invalidate the cache, if the changed code is `parse`d to the same expression as the previous version of the code. For example, if you have run `cache_rds({Sys.sleep(5);1+1})` before, running `cache_rds({ Sys.sleep( 5 ) ; 1 + 1 })` will use the cache, because the two expressions are essentially the same (they only differ in white spaces). Usually you can add/delete white spaces or comments to your code in `expr` without invalidating the cache. See the package vignette `vignette('xfun',package = 'xfun')` for more examples.

When this function is called in a code chunk of a **knitr** document, you may not want to provide the filename or directory of the cache file, because they have reasonable defaults.

Side-effects (such as plots or printed output) will not be cached. The cache only stores the last value of the expression in `expr`.

Examples

```
f = tempfile()  # the cache file
compute = function(...) {
    res = xfun::cache_rds({
        Sys.sleep(1)
        1:10
    }, file = f, dir = "", ...)
    res
}
compute()  # takes one second
compute()  # returns 1:10 immediately
compute()  # fast again
compute(rerun = TRUE)  # one second to rerun
compute()
file.remove(f)
```

`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.

---

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), ...)
```

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`).

Value

The integer code `0` for success, or an error if none of the methods work.

---

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 htmltools::a() (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().

These functions require R packages **mime** and **htmltools**. If you have installed the **rmarkdown** package, these packages should be available, otherwise you need to install them separately.

Currently Internet Explorer does not support downloading embedded files (https://caniuse.com/#feat=download). 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()) htmltools::browsable(link)
```

---

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)

sans_ext(x)

with_ext(x, ext)
```

Arguments

x	A character of file paths.
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")
```

---

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"))
```

`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

Read the HTML source of the release page and parse the tags of the releases.

Usage

```
github_releases(repo, subpath = "", pattern = "(v[0-9.]+)")
```

Arguments

repo	The repository name of the form user/repo, e.g., "yihui/xfun".
subpath	A character string to be appended to the URL of Github releases (i.e., https://github.com/user/repo/releases For example, you may use subpath = "latest" to get the tag of the latest release.
pattern	A regular expression to extract the tags from the HTML source. It must contain a group (i.e., must have a pair of parentheses).

Value

A character vector of (GIT) tags.

Examples

```
if (interactive()) xfun::github_releases("yihui/xfun")
```

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. This requires the **mime** package.

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)
```

---

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(src, build = TRUE, build_opts = NULL, install_opts = NULL)
```

Arguments

`src`	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()
})
```

isFALSE *Test if an object is identical to* FALSE

Description

A simple abbreviation of `identical(x,FALSE)`.

Usage

```
isFALSE(x)
```

Arguments

x	An R object.

Examples

```
library(xfun)
isFALSE(TRUE)  # false
isFALSE(FALSE)  # true
isFALSE(c(FALSE, FALSE))  # false
```

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 `xfun::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_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 `xfun::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()
```

Examples

```
library(xfun)
# only one of the following statements should be true
is_windows()
is_unix() && is_macos()
is_linux()
```

`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 xfun::in_dir() or use absolute paths.

Examples

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

---

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, windows_only = is_windows())
```

Arguments

x	A character vector.
windows_only	Whether to make the attempt on Windows only. On Unix, characters are typically encoded in the native encoding (UTF-8), so there is no need to do the conversion.

Examples

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

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

---

normalize_path *Normalize paths*

Description

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

Usage

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

Arguments

`x, winslash, must_work`
: Arguments passed to `normalizePath()`.

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. Values should be integers. 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)
```

---

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

Description

Calls the command optipng to optimize all PNG files under a directory.

Usage

```
optipng(dir = ".")
```

Arguments

dir	Path to a directory.

References

OptiPNG: http://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 expressions.

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)`.

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_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`.

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.

Examples

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

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

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.

Usage

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

Arguments

file	Path to a text file.
FUN	A function to process the text.
x	The content of the file.

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)
```

Arguments

x	A character vector of text in Markdown.

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 spaces 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. 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}"))
```

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_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, ...)
```

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()).

---

relative_path *Get the relative path of a path relative 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	The path to be converted to a relative path.
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 the path cannot be converted to a relative path.

Value

A relative path if the conversion succeeded; otherwise the original path 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")
```

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.

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 = FALSE, 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.
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. 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`'.

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(),
  ...,
  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.
`..., wait`	Arguments to be passed to `system2()`.
`fail`	The desired error message when an error occurred in calling the function.

Value

The returned value of the function in the new R session.

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))
```

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

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)
```

Arguments

`x`	A character vector of R source code.

Value

A list of character vectors, and each vector contains a complete R expression.

Examples

```
xfun::split_source(c("if (TRUE) {", "1 + 1", "}", "print(1:5)"))
```

`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!
```

---

stringsAsStrings *Set the global option* `options(stringsAsFactors = FALSE)` *inside a parent function and restore the option after the parent function exits*

Description

This is a shorthand of `opts = options(stringsAsFactors = FALSE); on.exit(options(opts), add = TRUE)`; `strings_please()` is an alias of `stringsAsStrings()`.

Usage

```
stringsAsStrings()

strings_please()
```

Examples

```
f = function() {
    xfun::strings_please()
    data.frame(x = letters[1:4], y = factor(letters[1:4]))
}
str(f())  # the first column should be character
```

---

tojson *A simple JSON serializer*

Description

A JSON serializer that only works on a limited types of R data (`NULL`, lists, logical scalars, character/numeric vectors). A character string of the class `JS_EVAL` 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.

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)
cat(tojson(list(a = 1, b = list(c = 1:3, d = "abc"))))
cat(tojson(list(c("a", "b"), 1:5, TRUE)))

# the class JS_EVAL is originally from htmlwidgets::JS()
JS = function(x) structure(x, class = "JS_EVAL")
cat(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_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

Run the command `curl -T file server` to upload a file to an FTP server. These functions require the system package (*not the R package*) `curl` to be installed (which should be available on macOS by default). 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,
  version = c("R-devel", "R-release", "R-oldrelease"),
  server = "ftp://win-builder.r-project.org/"
)
```

Arguments

`file`	Path to a local file.
`server`	The address of the FTP server.
`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. You may also consider using `devtools::check_win_*`, which currently only allows you to upload a package to one folder on win-builder each time, and `xfun::upload_win_builder()` uploads to all three folders, which is more likely to be what you need.

Value

Status code returned from `system2`.

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)
```

Arguments

x	A character vector of URLs.

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")
```

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)
```

Index