Package 'cmdfun'

October 12, 2022

Type Package

Title Framework for Building Interfaces to Shell Commands

Version 1.0.2

Description Writing interfaces to command line software is cumbersome.
'cmdfun' provides a framework for building function calls to seamlessly
interface with shell commands by allowing lazy evaluation of command line arguments.
'cmdfun' also provides methods for handling user-
specific paths to tool installs or secrets like API keys.
Its focus is to equally serve package builders who wish to wrap command line soft-
ware, and to help analysts stay inside
R when they might usually leave to execute non-R software.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Imports magrittr, purrr, R.utils, rlang, testthat, usethis, utils

RoxygenNote 7.0.2

Suggests cli, covr, knitr, processx, rmarkdown

VignetteBuilder knitr

URL `https://snystrom.github.io/cmdfun/`,
`https://github.com/snystrom/cmdfun`

BugReports `https://github.com/snystrom/cmdfun`

NeedsCompilation no

Author Spencer Nystrom [aut, cre, cph]
(<`https://orcid.org/0000-0003-1000-1579`>)

Maintainer Spencer Nystrom <`nystromdev@gmail.com`>

Repository CRAN

Date/Publication 2020-10-10 09:30:03 UTC

R topics documented:

`.check_valid_command_path` *Checks path is valid*

Description

Not meant to be called directly

Usage

```
.check_valid_command_path(path)
```

Arguments

`path`	path to file or directory

Value

expanded system path

Examples

```
if (.Platform$OS.type == "unix" & file.exists("~/bin")) {
# will return /full/path/to/home/bin, or error if path doesn't exist
.check_valid_command_path("~/bin")
}
```

`.check_valid_util` *Checks for valid members of subdirectory*

Description

Not meant to be called directly

Usage

```
.check_valid_util(util, utils = NULL, path = NULL)
```

Arguments

`util`	name of target located in path
`utils`	name of supported targets in path
`path`	path to directory

Value

safe path to util, or error if util does not exist

Examples

```
if (.Platform$OS.type == "unix") {
# this will return /full/path/to/bin
# or return an error for all values of util that are not "ls" and "pwd"
# or error if "ls" does not exist in "/bin"
.check_valid_util("ls", utils = c("ls", "pwd"), "/bin")

## Not run:
# This will throw error
.check_valid_util("badUtil", utils = c("ls", "pwd"), "/bin")

## End(Not run)
}
```

`cmd_args_all` *Return all named arguments and arguments passed as dots from parent function call*

Description

Return all named arguments and arguments passed as dots from parent function call

Usage

```
cmd_args_all(keep = NULL, drop = NULL)
```

Arguments

`keep`	name of arguments to keep
`drop`	name of arguments to drop (NOTE: keep or drop are mutually exclusive settings)

Value

named list of all arguments passed to parent

Examples

```
theFunction <- function(arg1, ...) { cmd_args_all() }
theArgs <-  theFunction(arg1 = "test", example = "hello")
```

`cmd_args_dots` *return function dots from parent function as named list*

Description

return function dots from parent function as named list

Usage

```
cmd_args_dots(keep = NULL, drop = NULL)
```

Arguments

`keep`	name of arguments to keep
`drop`	name of arguments to drop (NOTE: keep or drop are mutually exclusive settings)

Value

named list of kwargs from ...

Examples

```
theFunction <- function(...) { cmd_args_dots() }
theDots <-  theFunction(example = "hello", boolFlag = TRUE, vectorFlag = c(1,2,3))
```

`cmd_args_named` *Return all named arguments from parent function call*

Description

Return all named arguments from parent function call

Usage

```
cmd_args_named(keep = NULL, drop = NULL)
```

Arguments

`keep`	name of arguments to keep
`drop`	name of arguments to drop (NOTE: keep or drop are mutually exclusive settings)

Value

named list of all defined function arguments from parent

Examples

```
theFunction <- function(arg1, ...) { cmd_args_named() }
theNamedArgs <-  theFunction(arg1 = "test", example = "hello")
```

`cmd_error_if_missing` *Check that file(s) exist, error if not*

Description

Check that file(s) exist, error if not

Usage

```
cmd_error_if_missing(files)
```

Arguments

`files`	list or vector of paths to check

Value

nothing or error message for each missing file

Examples

```
cmd_error_if_missing(tempdir())
## Not run:
# Throws error if file doesn't exist
cmd_error_if_missing(file.path(tempdir(), "notreal"))

## End(Not run)
```

`cmd_file_combn` *Generates list of expected output files*

Description

See documentation of cmd_file_expect() for more details about how this works

Usage

```
cmd_file_combn(prefix, ext, outdir = ".")
```

Arguments

`prefix`	file name to be given each ext. If a character vector, must be equal length of ext or shorter
`ext`	file extension (no ".", ie "txt", "html")
`outdir`	optional directory where files should exist

Value

list of file paths by each ext or prefix (whichever is longer)

Examples

```
# Makes list for many file types of same prefix
# ie myFile.txt, myFile.html, myFile.xml
cmd_file_combn("myFile", c("txt", "html", "xml"))

# Makes list for many files of same type
# ie myFile1.txt, myFile2.txt, myFile3.txt
cmd_file_combn(c("myFile1", "myFile2", "myFile3"), "txt")
```

`cmd_file_expect` *Creates list of paths by file extension & checks they exist*

Description

Ext or prefix can be a vector or single character. The shorter value will be propagated across all values of the other. See Examples for details.

Usage

```
cmd_file_expect(prefix, ext, outdir = ".")
```

Arguments

`prefix`	name of file prefix for each extension.
`ext`	vector of file extensions
`outdir`	directory the files will be inside

Details

If files are not found, throws an error

Value

vector of valid file paths

Examples

```
## Not run:
# Expects many file types of same prefix
# ie myFile.txt, myFile.html, myFile.xml
cmd_file_expect("myFile", c("txt", "html", "xml"))

# Expects many files of same type
# ie myFile1.txt, myFile2.txt, myFile3.txt
cmd_file_expect(c("myFile1", "myFile2", "myFile3"), "txt")

# Expects many files with each prefix and each extension
# ie myFile1.txt, myFile1.html, myFile2.txt, myFile2.html
cmd_file_expect(c("myFile1", "myFile2"), c("txt", "html"))


## End(Not run)
```

cmd_help_flags_similar

Suggest alternative name by minimizing Levenshtein edit distance between valid and invalid arguments

Description

Suggest alternative name by minimizing Levenshtein edit distance between valid and invalid arguments

Usage

```
cmd_help_flags_similar(
  command_flag_names,
  flags,
  .fun = NULL,
  distance_cutoff = 3L
)
```

Arguments

`command_flag_names`	character vector of valid names (can be output of `cmd_help_parse_flags`)
`flags`	a vector names correspond to values to be checked against `command_flag_names`
`.fun`	optional function to apply to `command_flag_names` and `flags` before checking their values. If using a function to rename flags after `cmd_list_interp`, use that same function here. Can be useful for parsing help lines into R-friendly variable names for user-convenience. Can be function or `rlang`-style formula definition (ie `.fun = ~{foo(.x)}` is the same as `.fun = function(x){foo(x)}`). Note: if command_flag_names need additional parsing after `cmd_help_parse_flags`, it is best to do that preprocessing before passing them to this function.
`distance_cutoff`	Levenshtein edit distance beyond which to suggest ??? instead of most similar argument (default = 3). Setting this too liberally will result in nonsensical suggestions.

Value

named vector where names are names from `flags` and their values are the suggested best match from `command_flag_names`

Examples

```
# with a flagsList, need to pass names()
flagsList <- list("output" = "somevalue", "missplld" = "anotherValue")
cmd_help_flags_similar(c("output", "misspelled"), names(flagsList))
```

```
command_flags <- c("long-flag-name")
flags <- c("long_flag_naee")
cmd_help_flags_similar(command_flags, flags, .fun = ~{gsub("-", "_", .x)})

# returns NULL if no errors
cmd_help_flags_similar(c("test"), "test")
```

cmd_help_flags_suggest

Error & Suggest different flag name to user

Description

Error & Suggest different flag name to user

Usage

```
cmd_help_flags_suggest(suggest_names)
```

Arguments

`suggest_names`	named character vector, names correspond to original value, values correspond to suggested replacement.

Value

error message suggesting alternatives to user

Examples

```
user_flags <- list("output", "inpt")
valid_flags <- c("output", "input")
suggestions <- cmd_help_flags_similar(valid_flags, user_flags)
## Not run:
# Throws error
cmd_help_flags_suggest(suggestions)

## End(Not run)
```

cmd_help_parse_flags *Parses commandline help options to return vector of valid flag names*

Description

When using cmdfun to write lazy shell wrappers, the user can easily mistype a commandline flag since there is not text completion. Some programs behave unexpectedly when flags are typed incorrectly, and for this reason return uninformative error messages.

Usage

```
cmd_help_parse_flags(help_lines, split_newline = FALSE)
```

Arguments

`help_lines`	`character` vector containing the output of "command –help", or similar output. Optional: pass either `stdout`, or `stderr` output from processx::run(), must set `processx` = `TRUE`.
`split_newline`	`logical(1)` if set to TRUE will split string on "\n" before parsing (useful when parsing output from `processx`).

Details

`cmd_help_parse_flags` tries to grab flags from –help documentation which can be used for error checking. It will try to parse flags following "-" or "–" while ignoring hyphenated words in help text. Although this should cover most use-cases, it may be necessary to write a custom help-text parser for nonstandard tools. Inspect this output **carefully** before proceeding. Most often, characters are leftover at the **end** of parsed names, which will require additional parsing.

Value

character vector of flag names parsed from help text

See Also

`cmd_help_flags_similar` `cmd_help_flags_suggest`

Examples

```
if (.Platform$OS.type == "unix" & file.exists("/bin/tar")) {
# below are two examples parsing the --help method of GNU tar

# with processx
if (require(processx)) {
out <- processx::run("tar", "--help", error_on_status = FALSE)
fn_flags <- cmd_help_parse_flags(out$stdout, split_newline = TRUE)
}
```

```
# with system2
lines <- system2("tar", "--help", stderr = TRUE)
fn_flags <- cmd_help_parse_flags(lines)

# NOTE: some of the "tar" flags contain the extra characters: "\[", "\)", and ";"
# ie "one-top-level\[" which should be "one-top-level"
# These can be additionally parsed using
gsub("[\\[;\\)]", "", fn_flags)
}
```

`cmd_install_check` *Wrapper function for checking an install*

Description

This function can be lightly wrapped by package builders to build a user-friendly install checking function.

Usage

```
cmd_install_check(path_search, path = NULL)
```

Arguments

`path_search`	`function` output of `cmd_path_search()`
`path`	user-override path to check (identical to `path` argument of `cmd_path_search()` output)

Value

pretty printed message indicating whether files exits or not. Green check = Yes, red X = No.

Examples

```
## Not run:
path_search <- cmd_path_search(default = "/bin", utils = "ls")
cmd_install_check(path_search)

## End(Not run)
```

cmd_install_is_valid *Macro for constructing boolean check for valid path*

Description

Macro for constructing boolean check for valid path

Usage

```
cmd_install_is_valid(path_search, util = NULL)
```

Arguments

path_search	function output of `cmd_path_search()` **NOTE:** When passing the function, do not pass as: `fun()`, but `fun` to avoid evaluation.
util	value to pass to `util` argument of `path_search`, allows building individual functions for each util (if passing one of each), or for simultaneously checking all utils if setting `util` = `TRUE`. Will cause error if `util` = `TRUE` but no utils are defined. **NOTE:** There is no error checking for whether `util` is set correctly during the build process, so ensure correct spelling, etc. to avoid cryptic failures.

Value

a function returning TRUE or FALSE if a valid install is detected. With arguments: `path` (a path to install location), `util` an optional `character(1)` to

Examples

```
if (.Platform$OS.type == "unix") {
search <- cmd_path_search(option_name = "bin_path", default_path = "/bin/")
valid_install <- cmd_install_is_valid(search)
# Returns TRUE if "/bin/" exists
valid_install()
# Returns FALSE if "bad/path/" doesn't exist
valid_install("bad/path/")

# Also works with options
search_option_only <- cmd_path_search(option_name = "bin_path")
valid_install2 <- cmd_install_is_valid(search_option_only)
options(bin_path = "/bin/")
valid_install2()

# Setting util = TRUE will check that all utils are also installed
search_with_utils <- cmd_path_search(default_path = "/bin", utils = c("ls", "pwd"))
valid_install_all <- cmd_install_is_valid(search_with_utils, util = TRUE)
valid_install_all()
}
```

`cmd_list_drop` *Drop entries from list of flags by name, name/value pair, or index*

Description

Drop entries from list of flags by name, name/value pair, or index

Usage

```
cmd_list_drop(flags, drop)
```

Arguments

`flags`	named list output of cmd_list_interp
`drop`	vector of flag entries to drop. Pass a character vector to drop flags by name. Pass a named vector to drop flags by name/value pairs. Pass a numeric vector to drop by position.

Value

flags list with values in drop removed

Examples

```
exFlags <- list("flag1" = 2, "flag2" = "someText")
cmd_list_drop(exFlags, "flag1")
# will drop flag2 because its name and value match 'drop' vector
cmd_list_drop(exFlags, c("flag2" = "someText"))
# Will drop "flag1" by position index
cmd_list_drop(exFlags, 1)

# won't drop flag2 because its value isn't 'someText'
exFlags2 <- list("flag1" = 2, "flag2" = "otherText")
cmd_list_drop(exFlags, c("flag2" = "someText"))
```

`cmd_list_drop_named` *Drop items by name from list*

Description

A pipe-friendly wrapper around `list[!(names(list) %in% names)]` This function is slightly faster than using `cmd_list_drop()` to drop items by name.

Usage

```
cmd_list_drop_named(list, names)
```

Arguments

`list`	an R list
`names`	vector of names to drop

Value

list removing items defined by names

Examples

```
cmd_list_drop_named(list("a" = 1, "b" = 2), "a")
```

`cmd_list_interp` *Convert list of function arguments to list of command flags*

Description

Function also handles error checking to ensure args contain valid data types, and looks for common usage mistakes.

Usage

```
cmd_list_interp(args, flag_lookup = NULL)
```

Arguments

`args`	named list output from get*Args family of functions.
`flag_lookup`	optional named vector used to convert args to command flags

Details

The list structure is more amenable to manipulation by package developers for advanced use before evaluating them to the command flags vector with cmd_list_to_flags().

Value

named list

Examples

```
theFunction <- function(...){cmd_args_all()}
theArgs <- theFunction(arg1 = "value", arg2 = TRUE)
flagList <- cmd_list_interp(theArgs)
flags <- cmd_list_to_flags(flagList)
```

cmd_list_keep *keep entries from list of flags by name, name/value pair, or index*

Description

keep entries from list of flags by name, name/value pair, or index

Usage

```
cmd_list_keep(flags, keep)
```

Arguments

`flags`	named list output of cmd_list_interp
`keep`	vector of flag entries to keep. Pass a character vector to keep flags by name. Pass a named vector to keep flags by name/value pairs. Pass a numeric vector to keep by position.

Value

flags list with values not in keep removed

Examples

```
exFlags <- list("flag1" = 2, "flag2" = "someText")
cmd_list_keep(exFlags, "flag1")
# will keep flag2 because its name and value match 'keep' vector
cmd_list_keep(exFlags, c("flag2" = "someText"))
# Will keep "flag1" by position index
cmd_list_keep(exFlags, 1)

# won't keep flag2 because its value isn't 'someText'
exFlags2 <- list("flag1" = 2, "flag2" = "otherText")
cmd_list_keep(exFlags, c("flag2" = "someText"))
```

cmd_list_keep_named *Keep items by name from list*

Description

A pipe-friendly wrapper around `list[(names(list) %in% names]`.

Usage

```
cmd_list_keep_named(list, names)
```

Arguments

`list`	an R list
`names`	vector of names to keep

Details

This function is slightly faster than using `cmd_list_keep()` to keep items by name.

Value

list keeping only items defined by names

Examples

```
cmd_list_keep_named(list("a" = 1, "b" = 2), "a")
```

`cmd_list_to_flags` *Convert flag list to vector of command flags*

Description

Convert flag list to vector of command flags

Usage

```
cmd_list_to_flags(flagList, prefix = "-", sep = ",")
```

Arguments

`flagList`	output from cmd_list_interp(). A named list where names correspond to flags and members correspond to the value for the flag.
`prefix`	flag prefix, usually "-" or "–".
`sep`	separator to use if flag has a vector of values (default: NULL).

Value

character vector of parsed commandline flags followed by their values

Examples

```
theFunction <- function(...){cmd_args_all()}
theArgs <- theFunction(arg1 = "value", arg2 = TRUE)
flagList <- cmd_list_interp(theArgs)
flags <- cmd_list_to_flags(flagList)
```

cmd_path_search *Macro for constructing target path validators*

Description

A common pattern in designing shell interfaces is to ask the user to give an absolute path to the target shell utility. It is common to pass this information from the user to R by using either R environment variables defined in .Renviron, using options (set with option(), and got with getOption()), having the user explicitly pass the path in the function call, or failing this, using a default install path.

Usage

```
cmd_path_search(
  environment_var = NULL,
  option_name = NULL,
  default_path = NULL,
  utils = NULL
)
```

Arguments

`environment_var`	name of R environment variable defining target path. Can be set in .Renviron.
`option_name`	name of user-configurable option (called by getOption) which will hold path to target
`default_path`	default install path of target. Can contain shell specials like "~" which will be expanded at runtime (as opposed to build time of the search function).
`utils`	optional character vector containing names of valid utils inside target path, used to populate error checking for valid install.

Details

Another common use-case involves software packages with many tools packaged in a single directory, and the user may want to call one or many utilities within this common structure.

For example, the software "coolpackage" is installed in "~/coolpackage", and has two programs: "tool1", and "tool2" found in "~/coolpackage/tool1" and ~/coolpackage/tool2", respectively.

To design an interface to coolpackage, this function can automate checking and validation for not only the package, but for each desired utility in the package.

The hierarchy of path usage is: user-defined > option_name > environment_var > default_path

Value

function that returns a valid path to tool or optional utility.

The returned path_search function takes as input a path or util. where path is a user override path for the supported tool. If the user-defined path is invalid, this will always throw an error and not search the defined defaults.

util must be found within the target path, but does not have to be present in the original "utils" call. The user will be warned if this is the case. If `util` is set to `TRUE` will return all paths to utilities without checking the install. This can be used for writing user-facing install checkers.

Examples

```
if (.Platform$OS.type == "unix") {
bin_checker <- cmd_path_search(default_path = "/bin", utils = c("ls", "pwd"))
# returns path to bin
bin_checker()
# returns path to bin/ls
bin_checker(util = "ls")
}
```

`cmd_ui_file_exists` *Checks if file exists, returns pretty status message*

Description

Checks if file exists, returns pretty status message

Usage

```
cmd_ui_file_exists(file)
```

Arguments

`file`	path to file

Value

ui_done or ui_oops printed to terminal.

Examples

```
cmd_ui_file_exists("/path/to/file.txt")
```

Index