Package 'this.path'

September 3, 2022

Version 0.11.0

License MIT + file LICENSE

Title Get Executing Script's Path, from 'RStudio', 'Rgui', 'VSCode', 'Rscript' (Shells Including Windows Command-Line / / Unix Terminal), and 'source'

Description Determine the full path of the executing script. Works when running a line or selection from a script in 'RStudio', 'Rgui', and 'VSCode', when using 'source', 'sys.source', 'debugSource' in 'RStudio', and 'testthat::source_file', and when running from a shell.

Author Andrew Simmons

Maintainer Andrew Simmons <akwsimmo@gmail.com>

Suggests utils, microbenchmark

Enhances rprojroot, rstudioapi, testthat

URL https://github.com/ArcadeAntics/this.path

BugReports https://github.com/ArcadeAntics/this.path/issues

Encoding UTF-8

NeedsCompilation no

Repository CRAN

Date/Publication 2022-09-03 21:40:02 UTC

R topics documented:

this.path-package 2
Args 3
as.relative.path 5
check.path 6
here 7
R.from.shell 9
shFILE 11
this.path 12
this.path-deprecated 15

this.path.in.VSCode 16
this.proj 17
tryCatch2 17

Index **19**

`this.path-package` *Get Executing Script's Path, from 'RStudio', 'Rgui', 'VSCode', 'Rscript' (Shells Including Windows Command-Line / / Unix Terminal), and 'source'*

Description

Determine the full path of the executing script. Works when running a line or selection from a script in 'RStudio', 'Rgui', and 'VSCode', when using 'source', 'sys.source', 'debugSource' in 'RStudio', and 'testthat::source_file', and when running from a shell.

Details

The three most important functions from this package are `this.path`, `this.dir`, and `here`.

`this.path()` returns the `normalized` path of the executing script.

`this.dir()` returns the `normalized` path of the directory in which the executing script is located.

`here()` constructs file paths relative to the executing script's directory.

Note

This package started from a stack overflow posting, found at:

https://stackoverflow.com/questions/1815606/determine-path-of-the-executing-script

If you like this package, please consider upvoting my answer so that more people will see it! If you have an issue with this package, please use `utils::bug.report(package = "this.path")` to report your issue.

Author(s)

Andrew Simmons

Maintainer: Andrew Simmons <akwsimmo@gmail.com>

See Also

The main functions from **this.path**: `this.path`, `this.dir`, `here`

Check `this.path()` is functioning correctly: `check.path`, `check.dir`

Extract 'FILE' from command line arguments: `shFILE`, `normalized.shFILE`

`source`, `sys.source`, `debugSource`, `testthat::source_file`

`R.from.shell`

Args *Providing Arguments to a Script*

Description

withArgs allows you source R code while providing arguments. This would be in the circumstance that you want to run a script and provide command-line arguments, but want the objects to appear in your environment, or if you just don't want to shQuote the arguments.

fileArgs is a generalized version of commandArgs, allowing you to access the script's arguments whether it was sourced or run from a shell.

asArgs coerces R objects into a character vector, for use with command line applications and withArgs.

from.shell tells you if the R script was run from a shell.

Usage

```
asArgs(...)
fileArgs()
withArgs(expr, ...)

from.shell()
```

Arguments

...	R objects to turn into scripts arguments; typically logical, numeric, character, Date, and POSIXt vectors.
expr	an (unevaluated) call to source, sys.source, debugSource, or testthat::source_file.

Details

... is first put into a list, and then each non-list element is converted to character. They are converted as follows:

Factors (class "factor"**)** using as.character.factor

Date-Times (class "POSIXct" **and** "POSIXlt"**)** using format "%Y-%m-%d %H:%M:%OS6" (retains as much precision as possible)

Numbers (class "numeric" **and** "complex"**)** with 17 significant digits (retains as much precision as possible) and "." as the decimal point character.

Raw Bytes (class "raw"**)** using sprintf("0x%02x",) (can easily convert back to raw with as.raw() or as.vector(, "raw"))

All others will be converted to character using as.character and its methods.

The arguments will then be unlisted, and all attributes will be removed. Arguments that are NA_character_ after conversion will be converted to "NA" (since the command-line arguments also never have missing strings).

Consider that it may be better to use `Rscript` combined with `save` and `load` or `saveRDS` and `readRDS`.

Also consider that what you want is a function, and not a script. If you're already at the R level, it is easier and more flexible to source a script that creates a function, and then use that function. This only applies if the script you are sourcing will never be run from the command-line, or at least will never be run on its own (it will only be used in the context of other scripts, it will *NEVER* be used as a stand alone script).

Value

for `asArgs` and `fileArgs`, a character vector.

for `withArgs`, the result of evaluating `expr`.

Examples

```
asArgs(
    NULL,
    c(TRUE, FALSE, NA),
    1:5,
    pi,
    exp(6i),
    letters[1:5],
    as.raw(0:4),
    as.Date("1970-01-01"),
    as.POSIXct("1970-01-01 00:00:00"),
    list(
        list(
            list(
                "lists are recursed"
            )
        )
    )
)


this.path:::write.code(file = FILE <- tempfile(), {
    withAutoprint({


        this.path()
        fileArgs()
        from.shell()


    }, verbose = FALSE)
})


# wrap your source call with a call to 'withArgs'
withArgs(
    source(FILE, local = TRUE, verbose = FALSE),
```

```
    letters, pi, exp(1)
)
withArgs(
    sys.source(FILE, environment()),
    letters, pi + 1i * exp(1)
)
this.path:::.Rscript(c("--default-packages=this.path", "--vanilla", FILE,
    asArgs(letters, pi, as.POSIXct("2022-07-17 04:25"))))


# with R >= 4.1.0, use the forward pipe operator '|>' to
# make calls to 'withArgs' more intuitive:
# source(FILE, local = TRUE, verbose = FALSE) |> withArgs(
#     letters, pi, exp(1)
# )
# sys.source(FILE, environment()) |> withArgs(
#     letters, pi + 1i * exp(1)
# )
```

`as.relative.path` *Make a Path Relative to Another Path*

Description

When working with **this.path**, you will be dealing with a lot of absolute paths. These paths are no good for saving within files, so you'll need to use `as.relative.path()` to turn your absolute paths into relative paths.

Usage

```
as.relative.path(path, relative.to = this.dir(verbose = FALSE))
as.rel.path     (path, relative.to = this.dir(verbose = FALSE))
```

Arguments

`path`	character vector of file / / URL paths.
`relative.to`	character string; the file / / URL path to make `path` relative to.

Details

Tilde-expansion (see `path.expand`) is first done on `path` and `relative.to`.

If `path` and `relative.to` are equivalent, `"."` will be returned. If `path` and `relative.to` have no base in common, the `normalized` `path` will be returned.

Value

character vector of the same length as `path`.

Examples

```
# Windows example


# as.relative.path(
#     c(
#         # paths which are equivalent will return "."
#         "C:/Users/effective_user/Documents/this.path/man",
#
#
#         # paths which have no base in common return as themselves
#      "https://raw.githubusercontent.com/ArcadeAntics/this.path/main/tests/this.path_w_URLs.R",
#         "D:/",
#         "//host-name/share-name/path/to/file",
#
#
#         "C:/Users/effective_user/Documents/testing",
#         "C:\\Users\\effective_user",
#         "C:/Users/effective_user/Documents/R/this.path.R"
#     ),
#     relative.to = "C:/Users/effective_user/Documents/this.path/man"
# )


# Unix-alikes example


# as.relative.path(
#     c(
#         # paths which are equivalent will return "."
#         "/home/effective_user/Documents/this.path/man",
#
#
#         # paths which have no base in common return as themselves
#      "https://raw.githubusercontent.com/ArcadeAntics/this.path/main/tests/this.path_w_URLs.R",
#         "//host-name/share-name/path/to/file",
#
#
#         "/home/effective_user/Documents/testing",
#         "/home/effective_user",
#         "/home/effective_user/Documents/R/this.path.R"
#     ),
#     relative.to = "/home/effective_user/Documents/this.path/man"
# )
```

---

check.path *Check* this.path() *is Functioning Correctly*

Description

Add `check.path("path/to/file")` to the beginning of your script to initialize `this.path()`, and check that `this.path()` is returning the path you expect.

Usage

```
check.path(...)
check.dir(...)
```

Arguments

`...`	further arguments passed to `file.path` which must return a character string; the path you expect `this.path()` or `this.dir()` to return. The specified path can be as deep as necessary (just the basename, the last directory name and the basename, the last two directory names and the basename, ...), but using an absolute path is not intended (recommended against). `this.path()` makes R scripts portable, but using an absolute path in `check.path` or `check.dir` makes an R script non-portable, defeating the whole purpose of this package.

Value

If the expected path / / directory matches `this.path()` / / `this.dir()`, then `TRUE` invisibly.

Otherwise, an error is raised.

Examples

```
# I have a project called 'EOAdjusted'
#
# Within this project, I have a folder called 'code'
# where I place all of my scripts.
#
# One of these scripts is called 'provrun.R'
#
# So, at the top of that R script, I could write:


# this.path::check.path("EOAdjusted", "code", "provrun.R")
#
# or
#
# this.path::check.path("EOAdjusted/code/provrun.R")
```

`here` *Construct Path to File, Beginning with* `this.dir()`

Description

Construct the path to a file from components in a platform-independent way, starting with `this.dir()`.

Usage

```
here(..., .. = 0)
ici(..., .. = 0)
```

Arguments

`...`	further arguments passed to `file.path()`.
`..`	the number of directories to go back.

Details

The path to a file begins with a base. The base is `..` number of directories back from the executing script's directory (`this.dir()`). The argument is named `..` because `".."` refers to the parent directory in Windows, Unix, and URL paths alike.

Value

A character vector of the arguments concatenated term-by-term, beginning with the executing script's directory.

Examples

```
this.path:::write.code(file = FILE <- tempfile(), {


    this.path::here()
    this.path::here(.. = 1)
    this.path::here(.. = 2)


    # use 'here' to read input from a file located nearby
    this.path::here(.. = 1, "input", "file1.csv")


    # or maybe to run another script
    this.path::here("script2.R")


})


source(FILE, echo = TRUE, verbose = FALSE)
```

R.from.shell *Using R From a Shell*

Description

How to use R from a shell (including the Windows command-line / / Unix terminal).

Details

For the purpose of running R scripts, there are four ways to do it. Suppose our R script has filename 'script1.R', we could write any of:

```
R -f script1.R
R --file=script1.R
R CMD BATCH script1.R
Rscript script1.R
```

The first two are different ways of writing equivalent statements. The third statement is the first statement plus options '--restore' '--save' (plus option '--no-readline' under Unix-alikes), and it also saves the stdout and stderr in a file of your choosing. The fourth statement is the second statement plus options '--no-echo' '--no-restore'. You can try:

```
R --help
R CMD BATCH --help
Rscript --help
```

for a help message that describes what these options mean. In general, Rscript is the one you want to use. It should be noted that Rscript has some exclusive environment variables (not used by the other executables) that will make its behaviour different from R.

For the purpose of making packages, R CMD is what you'll need to use. Most commonly, you'll use:

```
R CMD build
R CMD INSTALL
R CMD check
```

R CMD build will turn an R package (specified by a directory) into tarball. This allows for easy sharing of R packages with other people, including submitting a package to CRAN . R CMD INSTALL will install an R package (specified by a directory or tarball), and is used by utils::install.packages. R CMD check will check an R package (specified by a tarball) for possible errors in code, documentation, tests, and much more.

If, when you execute one of the previous commands, you see the following error message: "'R' is not recognized as an internal or external command, operable program or batch file.", see section **Ease of Use on Windows**.

Ease of Use on Windows

Under Unix-alikes, it is easy to invoke an R session from a shell by typing the name of the R executable you wish to run. On Windows, you should see that typing the name of the R executable you wish to run does not run that application, but instead signals an error. Instead, you will have to type the full path of the directory where your R executables are located (see section **Where are my R executable files located?**), followed by the name of the R executable you wish to run.

This is not very convenient to type everytime something needs to be run from a shell, plus it has another issue of being computer dependent. The solution is to add the path of the directory where your R executables are located to the `Path` environment variable. The `Path` environment variable is a list of directories where executable programs are located. When you type the name of an executable program you wish to run, Windows looks for that program through each directory in the `Path` environment variable. When you add the full path of the directory where your R executables are located to your `Path` environment variable, you should be able to run any of those executable programs by their basenames ('R', 'Rcmd', 'Rscript', and 'Rterm') instead of their full paths.

To add a new path to your `Path` environment variable:

1. Open the **Control Panel**
2. Open category **User Accounts**
3. Open category **User Accounts** (again)
4. Open **Change my environment variables**
5. Click the variable `Path`
6. Click the button **Edit...**
7. Click the button **New**
8. Type (or paste) the full path of the directory where your R executables are located, and press **OK**

This will modify your environment variable `Path`, not the systems. If another user wishes to run R from a shell, they will have to add the directory to their `Path` environment variable as well.

If you wish to modify the system environment variable `Path` (you will need admin permissions):

1. Open the **Control Panel**
2. Open category **System and Security**
3. Open category **System**
4. Open **Advanced system settings**
5. Click the button **Environment Variables...**
6. Modify `Path` same as before, just select `Path` in **System variables** instead of **User variables**

To check that this worked correctly, open a shell and execute the following commands:

```
R --help
R --version
```

You should see that the first prints the usage message for the R executable while the second prints information about the version of R currently being run. If you have multiple versions of R installed, make sure this is the version of R you wish to run.

Where are my R executable files located?

In an R session, you can find the location of your R executable files with the following command:

```
cat(sQuote(normalizePath(R.home("bin"))), "\n")
```

On Windows, for me, this is:

'`C:\Program Files\R\R-4.2.0\bin\x64`'

On Linux, for me, this is:

'`/usr/lib/R/bin`'

shFILE *Get Argument 'FILE' Provided to R by a Shell*

Description

Look through the command line arguments, extracting 'FILE' from either of the following: '--file=FILE' or '-f' 'FILE'

Usage

```
shFILE(default, else.)
normalized.shFILE(default, else.)
```

Arguments

default	if 'FILE' was not found, this value is returned.
else.	missing or a function to call on the return value if 'FILE' is found. See tryCatch2 for inspiration.

Value

character string, or default if the command line argument 'FILE' was not found.

Note

Both functions will save their return values; this makes them faster when called subsequent times.

For normalized.shFILE, the path on Windows will use / as the file separator.

See Also

this.path, here

Examples

```
this.path:::write.code(file = FILE <- tempfile(), {

    withAutoprint({

        shFILE()
        normalized.shFILE()
        normalized.shFILE(default = {
            stop("interestingly enough, because 'FILE' will be found,\n",
                "  argument 'default' won't be evaluated, and so this\n",
                "  error won't actually print, isn't that neat? you can\n",
                "  use this to your advantage in a similar manner, doing\n",
                "  arbitrary things only if 'FILE' isn't found")
        })
```

```
    }, width.cutoff = 60L, verbose = FALSE)

})
this.path:::.Rscript(c("--default-packages=this.path", "--vanilla", FILE))


for (expr in c("shFILE()",
               "shFILE(default = NULL)",
               "normalized.shFILE()",
               "normalized.shFILE(default = NULL)"))
    this.path:::.Rscript(c("--default-packages=this.path", "--vanilla", "-e", expr))
```

`this.path` *Determine Executing Script's Filename*

Description

`this.path()` returns the `normalized` path of the executing script.

`this.dir()` returns the `normalized` path of the directory in which the executing script is located.

`this.path2()`, `this.dir2()`, and `this.dir3()` are from an old release that did not include `default` as an argument in `this.path()` or `this.dir()`. They should not be used in future code development and should be removed from older code.

Usage

```
this.path(verbose = getOption("verbose"), default, else.)
this.dir(..., default, else.)

this.path2(...)  # deprecated, use this.path(..., default = NULL) instead
this.dir2(...)   # deprecated, use this.dir(..., default = NULL) instead

this.dir3(...)   # deprecated, use this.dir(..., default = getwd()) instead
```

Arguments

`verbose`	`TRUE` or `FALSE`; should the method in which the path was determined be printed?
`default`	if there is no executing script, this value is returned.
`else.`	missing or a function to call on the return value if there is an executing script. See `tryCatch2` for inspiration.
`...`	arguments passed to `this.path`.

Details

There are three ways in which R code is typically run:

1. in 'RStudio' / / 'Rgui' / / 'VSCode' by running the current line / / selection with the **Run** button / / appropriate keyboard shortcut
2. through a source call: a call to function `source`, `sys.source`, `debugSource` in 'RStudio', or `testthat::source_file`
3. from a shell, such as the Windows command-line / / Unix terminal

If you are using `this.path` in 'VSCode', see `this.path.in.VSCode`

To retrieve the executing script's filename, first an attempt is made to find a source call. The calls are searched in reverse order so as to grab the most recent source call in the case of nested source calls. If a source call was found, the argument *file* (*fileName* in the case of `debugSource`, *path* in the case of `testthat::source_file`) is returned from the function's evaluation environment. If you have your own source-like function that you'd like to be recognized by `this.path`, please contact the package maintainer so that it can be implemented.

If no source call is found up the calling stack, then an attempt is made to figure out how R is currently being used.

If R is being run from a shell, the shell arguments are searched for '`-f`' '`FILE`' or '`--file=FILE`' (the two methods of taking input from '`FILE`'). If exactly one of either type of argument is supplied, the text '`FILE`' is returned. It is an error to use `this.path` when none or multiple arguments of either type are supplied.

If R is being run from a shell under Unix-alikes with '`-g`' '`Tk`' or '`--gui=Tk`', `this.path()` will signal an error. This is because 'Tk' does not make use of its '`-f`' '`FILE`', '`--file=FILE`' argument.

If R is being run from 'RStudio', the active document's filename (the document in which the cursor is active) is returned (at the time of evaluation). If the active document is the R console, the source document's filename (the document open in the current tab) is returned (at the time of evaluation). Please note that the source document will *NEVER* be a document open in another window (with the **Show in new window** button). It is important to not leave the current tab (either by closing or switching tabs) while any calls to `this.path` have yet to be evaluated in the run selection. It is an error for no documents to be open or for a document to not exist (not saved anywhere).

If R is being run 'VSCode', the source document's filename is returned (at the time of evaluation). It is important to not leave the current tab (either by closing or switching tabs) while any calls to `this.path` have yet to be evaluated in the run selection. It is an error for a document to not exist (not saved anywhere).

If R is being run from 'Rgui', the source document's filename (the document most recently interacted with besides the R Console) is returned (at the time of evaluation). Please note that minimized documents will be *INCLUDED* when looking for the most recently used document. It is important to not leave the current document (either by closing the document or interacting with another document) while any calls to `this.path` have yet to be evaluated in the run selection. It is an error for no documents to be open or for a document to not exist (not saved anywhere).

If R is being run from 'AQUA', the executing script's path cannot be determined. Unlike 'RStudio' and 'Rgui', there is currently no way to request the path of an open document. Until such a time that there is a method for requesting the path of an open document, consider using 'RStudio' or 'VSCode'.

If R is being run in another manner, it is an error to use `this.path`.

If your GUI of choice is not implemented with `this.path`, please contact the package maintainer so that it can be implemented.

Value

character string; the executing script's filename.

Note

The first time `this.path` is called within a script, it will `normalize` the script's path, check that the script exists (throwing an error if it does not), and save it in the appropriate environment. When `this.path` is called subsequent times within the same script, it returns the saved path. This will be faster than the first time, will not check for file existence, and will be independent of the working directory.

As a side effect, this means that a script can delete itself using `file.remove` or `unlink` but still know its own path for the remainder of the script.

Within a script that contains calls to both `this.path` and `setwd`, `this.path` *MUST* be used *AT LEAST* once before the first call to `setwd`. This isn't always necessary; for instance if you ran a script using its absolute path as opposed to its relative path, changing the working directory has no effect. However, it is still advised against.

The following is *NOT* an example of bad practice:

```
setwd(this.path::this.dir())
```

`setwd` is most certainly written before `this.path()`, but `this.path()` will be evaluated first. It is not the written order that is bad practice, but the order of evaluation. Do not change the working directory before calling `this.path` at least once.

See Also

`here`

`shFILE`

`this.path-package`

`source`, `sys.source`, `debugSource`, `testthat::source_file`

`R.from.shell`

Examples

```
this.path:::write.code(file = FILE <- tempfile(), {

    withAutoprint({

        cat(sQuote(this.path::this.path(verbose = TRUE, default = {
            stop("interestingly enough, because the executing script's\n",
                "  path will be found, argument 'default' won't be evaluated,\n",
                "  and so this error won't actually print, isn't that\n",
                "  neat? you can use this to your advantage in a similar\n",
```

```
                "  manner, doing arbitrary things only if the executing\n",
                "  script does not exist")
        })), "\n\n")

    }, width.cutoff = 60L, verbose = FALSE)

})


source(FILE, verbose = FALSE)
sys.source(FILE, envir = environment())
if (.Platform$GUI == "RStudio")
    get("debugSource", "tools:rstudio", inherits = FALSE)(FILE)
if (requireNamespace("testthat"))
    testthat::source_file(FILE, chdir = FALSE, wrap = FALSE)


this.path:::.Rscript(c("--default-packages=NULL", "--vanilla", FILE))


# this.path also works when source-ing a URL
# (included tryCatch in case an internet connection is not available)
tryCatch({
   source("https://raw.githubusercontent.com/ArcadeAntics/this.path/main/tests/this.path_w_URLs.R")
}, condition = message)


for (expr in c("this.path()",
               "this.path(default = NULL)",
               "this.dir()",
               "this.dir(default = NULL)",
               "this.dir(default = getwd())"))
    this.path:::.Rscript(c("--default-packages=this.path", "--vanilla", "-e", expr))


# an example from R package 'logr'
this.path::this.path(verbose = FALSE, default = "script.log",
    else. = function(x) {
        x <- if (.Platform$OS.type == "windows")
            sub("[.][^.\\/]+$", "", x)
        else sub("[.][^./]+$", "", x)
        paste0(x, ".log")
    })
```

`this.path-deprecated` *Deprecated Functions in Package* **this.path**

Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as the next release.

Usage

```
this.path2(...)
this.dir2(...)

this.dir3(...)
```

Arguments

...	arguments passed to this.path.

See Also

this.path2-deprecated

this.dir2-deprecated

this.dir3-deprecated

---

this.path.in.VSCode this.path() *in 'VSCode'*

Description

this.path() will not work with a fresh installation of 'VSCode', some other packages / / applications are needed.

Details

You will need:

1. to install R packages **jsonlite** and **rlang**. On platforms without pre-built binaries, these packages will need compilation, you need to have make, gcc, and g++ installed. They can be installed from the terminal like:

   sudo apt 'install' 'make'

   sudo apt 'install' 'gcc'

   sudo apt 'install' 'g++'

2. to install R package **rstudioapi**. This package does not require compilation. Note that you do not need to install 'RStudio'.

3. to install the R extension for 'VSCode'; a prompt to install it will appear upon opening an R script in 'VSCode'.

this.proj *Construct Path to File, Beginning with Your Project Directory*

Description

`this.proj` behaves very similarly to `here::here` except that you can have multiple projects in use at once, and it will choose which project directory is appropriate based on `this.dir()`. Arguably, this makes it better than `here::here`.

Usage

```
this.proj(...)
```

Arguments

`...`	further arguments passed to `file.path()`.

Value

A character vector of the arguments concatenated term-by-term, beginning with the project directory.

See Also

`here`

tryCatch2 *Condition Handling and Recovery*

Description

A variant of `tryCatch` that accepts an `else.` argument, similar to `try except` in 'Python'.

Usage

```
tryCatch2(expr, ..., else., finally)
```

Arguments

`expr`	expression to be evaluated.
`...`	condition handlers.
`else.`	expression to be evaluated if evaluating `expr` does not throw an error nor a condition is caught.
`finally`	expression to be evaluated before returning or exiting.

Details

The use of the `else.` argument is better than adding additional code to `expr` because it avoids accidentally catching a condition that wasn't being protected by the `tryCatch` call.

Examples

```
FILES <- tempfile(c("existent-file_", "non-existent-file_"))
writeLines("line1\nline2", FILES[[1L]])
for (FILE in FILES) {
    con <- file(FILE)
    tryCatch2({
        open(con, "r")
    }, condition = function(cond) {
        cat("cannot open", FILE, "\n")
    }, else. = {
        cat(FILE, "has", length(readLines(con)), "lines\n")
    }, finally = {
        close(con)
    })
}
unlink(FILES)
```

Index

∗ **package**
    this.path-package, 2

Args, 3
as.character, *3*
as.character.factor, *3*
as.raw, *3*
as.rel.path *(*as.relative.path*)*, 5
as.relative.path, 5
as.vector, *3*
asArgs *(*Args*)*, 3

character, *3*
check.dir, *2*
check.dir *(*check.path*)*, 6
check.path, *2*, 6
commandArgs, *3*

Date, *3*

file.path, *7*, *8*, *17*
file.remove, *14*
fileArgs *(*Args*)*, 3
from.shell *(*Args*)*, 3

here, *2*, 7, *11*, *14*, *17*

ici *(*here*)*, 7

load, *4*
logical, *3*

normalize, *14*
normalized, *2*, *5*, *12*
normalized.shFILE, *2*
normalized.shFILE *(*shFILE*)*, 11
numeric, *3*

path.expand, *5*
POSIXt, *3*

R.from.shell, *2*, 9, *14*
readRDS, *4*
Rscript, *4*

save, *4*
saveRDS, *4*
setwd, *14*
shFILE, *2*, 11, *14*
shQuote, *3*
source, *2*, *3*, *13*, *14*
sprintf, *3*
stderr, *9*
stdout, *9*
sys.source, *2*, *3*, *13*, *14*

testthat::source_file, *2*, *3*, *13*, *14*
this.dir, *2*, *7*, *17*
this.dir *(*this.path*)*, 12
this.dir2 *(*this.path-deprecated*)*, 15
this.dir2-deprecated *(*this.path*)*, 12
this.dir3 *(*this.path-deprecated*)*, 15
this.dir3-deprecated *(*this.path*)*, 12
this.path, *2*, *11*, 12, *16*
this.path-deprecated, 15
this.path-package, 2
this.path.in.VSCode, *13*, 16
this.path2 *(*this.path-deprecated*)*, 15
this.path2-deprecated *(*this.path*)*, 12
this.proj, 17
tryCatch, *17*
tryCatch2, *11*, *12*, 17

unlink, *14*
utils::bug.report, *2*
utils::install.packages, *9*

withArgs *(*Args*)*, 3