The R language

Chris Johnson

12/4/22

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 {base}

`is.atomic()` if numeric, logical, character, complex, raw, or null `is.language()` `is.recursive()` if list; returns FALSE if S4

`show()` for console; `View()` for IDE. (Neither `return()` a value.)

```
random_sample <-
  sample(
    x = letters,
    size = 10
  ) %>%
  show(
    object = .
  )
```

1.1 do.call()

```
do.call(
  what = function_name,
  args = list(arg_1, arg_2, arg_3)
)
```

is equivalent to

```
function_name(arg_1, arg_2, arg_3)
```

Ah! Leah! by Donnie Iris

1.2 Quoting functions (`{base}`)

Unlike `{rlang}` analogs, quoting functions from `{base}` do not support unquoting. (This was the motivation for quasiquotation.)

- `quote()`
- `substitute()`
- `alist()`
- `bquote()`
- `~`

1.3 Accessors

`::` is an accessor, and it used to access exported functions from a package.

`:::` is used to access non-exported functions. Non-exported functions may not be documented.

2 Dates, times, and datetimes

```r
library(readxl)
library(dplyr)
library(lubridate)
```

2.1 Problem

Occasionally, dates will be read in as a number representing some time since the origin.

The origin depends on how the data were recorded. Here are a few common origins:

Source	Origin
Excel	1899-12-30
Unix	1970-01-01

The origin

2.2 Solution

1. Determine the origin. E.g., in Excel, this is 1899-12-30.
2. If just interested in a date, then your variable needs to be numeric, and represent the number of days since the origin; if interested in a datetime, then your variable needs to be numeric, and represent the number of seconds since the origin.

2.2.1 Example

Suppose

- we have an Excel workbook named `metadata.xlsx` with a column named `timestamp`, and the dates look like we expect (e.g., `"2018-03-15 10:15"`)
- we read in `timestamp.xlsx` in R using `readxl::read_excel()`

- we're suprised to find that the values under `timestamp` don't look like dates (e.g., `"2018-03-15 10:15"` has become `"43173.666666666664"`)

What happened?

To simulate this scenario, here's a data set containing

- `what_we_got` (the time since the origin)
- `what_we_expected` (the timestamp we're used to [the target for this exercise])

```
metadata <-
  read_excel(
    path =
      file.path(
        "path/to/",
        "metadata.xlsx"
      ),
    sheet = "Timestamp"
  )
```

We want `what_we_got` (a number representing the time since the origin) to be `what_we_expected` (a timestamp that is formatted in a familiar way).

We don't know what units `what_we_got` is in, but we can easily find that out by

- determining the origin
- doing some simple math

E.g., the first value is `metadata$what_we_got[1]`. The origin for these values is 1899-12-30 (`%Y%m%d`).

If we assume this value is the number of days since the origin, and we just want the date, all we need to do is ensure we convert `what_we_got` to `numeric`, and use `lubridate::as_date()` with `origin = "1899-12-30 UTC"`:

```
metadata %>%
mutate(
  .data = .,
  what_we_got = as.numeric(what_we_got),
  what_we_got = as_date(what_we_got, origin = "1899-12-30 UTC")
)
```

To recover `"2018-03-15 10:15"` (the datetime), we need to convert `what_we_got` to a more granular unit: seconds.

To do so, convert

- days to hours (24 hours per day);
- hours to minutes (60 minutes per hour); and
- minutes to seconds (60 seconds per minute)

and use `lubridate::as_datetime()`:

```
metadata %>%
mutate(
  .data = .,
  what_we_got = as.numeric(what_we_got) * 24 * 60 * 60,
  what_we_got = as_datetime(what_we_got, origin = "1899-12-30 UTC")
)
```

That's it!

3 Debugging

https://rstudio-education.github.io/hopr/debug.html

After `browser()` has been called, submitting

- `c` or `cont` to exit the browser and continue execution
- `f` to finish execution of the current loop or function
- `s` to evaluate the next statement
- `where` to print a stack trace
- `r` to resume
- `Q` to quit

`help` can be submitted to see the above commands

If an object named `c`, `cont`, `f`, `s`, `where`, `r`, or `Q` exists, those must be wrapped with `get()`, e.g. `get("f")`.

4 `{methods}`

`S4`

Class definition and object construction occur at runtime. (In other languages, class definition occurs at compile-time, and object construction occurs at runtime)

inheritance	dispatch
multiple	multiple

slot

classes generics methods

prototypes constructors helpers validators

accessor functions

method dispatch multiple inheritance multiple dispatch

S3 and S4 interaction

setClass() to create class setGeneric() to create generic setMethod() to create method

S4 functions are defined in `{methods}`

methods::setClass(Class = "name", slots = c()) registers a class definition in a hidden global variable

methods::new(Class, key = "value")

methods::is(object) to introspect (inheritance)

If `r` is a `RasterLayer` object, try

```
class(r)
methods::is(r)
```

`@` generally should only be used in method definitions Use accessor functions if working with an S4 object defined by someone else Accessors are S4 generics

Set

```
setGeneric(name = "property", def = function(x) standardGeneric(f = "property"))
```

Get

```
setGeneric(name = "property<-", def = function(x, value) standardGeneric(f = "property<-"))
```

Define methods

```
setMethod(f = "property", signature = "name", function(x) x@property)

setMethod(f = "property<-", signature = "name", function(x, value) x@property <- value)
```

`{sloop}` has useful functions for finding S4 objects:

- `sloop::otype()`
- `sloop::ftype()`

Community agrees to use UpperCamelCase for class names

```
setClass(
  Class = "car",
  slots =
    c(
      make = "character",
      model = "character",
      year = "numeric",
      transmission = "character",
      n_doors = "numeric",
      mpg = "numeric"
    ),
  prototype =
    list(
      make = "NA_character_",
      model = "NA_character_",
      year = NA_integer_,
      transmission = NA_character_,
      n_doors = NA_integer_,
      mpg = NA_real_
    )
)
```

```
mustang <-
  methods::new(
    make = "Ford,
    model = "Mustang",
    year = 2004,
    transmission = "manual",
    n_doors = 2,
    mpg = 28.3
  )
```

Inheritance basically means a class can be built from subclasses:

```
setClass(
  Class = "driver",
  contains = "car",
  slots = c(vehicle = "car"),
  prototype = list(methods::new(Class = "car"))
)
```

Some functions in `{methods}` are intended to be used only by the developer, and not the user:

Define a validitor for the class which runs when the constructor runs:

```
setValidity(
  Class = "name",
  method = function()
)
```

Check validity of instantiated objects with `validObject()`

4.1 Generics

setGeneric(name = “name”, def = function(standardGeneric))

5 Operators

Some operators have functional forms. E.g.

- `<-` is the same as `assign()`
- `::` is the same as `getExportedValue()`

Operators also have a *prefix forms*. E.g.

- `x <- 3` is the same as ‘`` <-`(x, 3) ``
- `dplyr::mutate` is the same as ‘`` ::`(dplyr, mutate) ``
- `mtcars$mpg` is the same as ‘`` $`(mtcars, mpg) ``
- ```` mtcars[["mpg"]] is the same as ```[[````(mtcars, “mpg”)“

Note: `$` can take `mpg`, whereas `[[` requires `"mpg"`.

6 R6

6.1 Methods

Use ProperCase when naming the class generator.

```r
MyClass <- R6::R6Class(
  classname = "MyClass",
  public =
    list(
      dataset = NULL
    )
)
```

Classes are built similar to how scripts are ran. We can take advantage of this.

Suppose we setup the following directory structure:

```
classes/
  oatsClass/
```

Suppose we save the definition for `oatsClass` in `oatsClass.R` in `classes/oatsClass/`

```
classes/
  oatsClass/
    oatsClass.R
```

Currently, `oatsClass` has no methods.

We could define public methods, store them inside of a list, and pass those to the `public` argument of the call to `R6::R6Class()`. The skeleton would look like this…

```r
MyClass <- R6::R6Class(
  classname = "MyClass",
  public = list()
)
```

...and a concrete example would look like this:

```r
MyClass <- R6::R6Class(
  classname = "MyClass",
  public =
    list(
      dataset = NULL,
      initialize = function(path_to_csv) {
        self$dataset <-
          read.csv(
            file = path_to_csv,
            stringsAsFactors = FALSE
          )
      }
    )
)
```

We could put more method definitions in the list. If the number of methods grows large, we might consider the ideas of superclassing and subclassing, however even if the number of methods is small but the method definitions are very large, we might want to organize methods into separate files.

Above, I stated

> Classes are built similar to how scripts are ran. We can take advantage of this.

Our directory structure looks like this

```
classes/
  oatsClass/
    oatsClass.R
```

We may wish to add an initialization method in `initialize.R` and save it in `classes/oatsClass/` alongside `oatsClass.R`:

```
classes/
  oatsClass/
    oatsClass.R
```

Recall the initialization method was defined as

```
initialize = function(path_to_csv) {
  self$dataset <-
    read.csv(
      file = path_to_csv,
      stringsAsFactors = FALSE
    )
}
```

Suppose `initialize.R` contains the same definition:

```
initialize <- function(path_to_csv) {
  self$dataset <-
    read.csv(
      file = path_to_csv,
      stringsAsFactors = FALSE
    )
}
```

Note: Inside a list, we used `=` for assignment. If defining the method externally (i.e. outside of the class defintion), we can (and should) use `<-`.

We save `initialize.R` in `classes/oatsClass/`:

```
classes/
  oatsClass/
    oatsClass.R
    initialize.R
```

Recall our initial development of `oatsClass.R`:

```
MyClass <- R6::R6Class(
  classname = "MyClass",
  public =
    list(
      dataset = NULL
    )
)
```

Recall that classes are constructed in a similar way to how scripts are ran. If we ran this class definition, we would find that our class generator already has some methods that were created when we created the generator. One of those methods is `set()`.

`oatsClass$set()` has four formal arguments:

- `which`
- `name`
- `value`
- `overwrite` (has default argument `FALSE`)

`$set()` allows us to assign members to our class generator.

> We can take advantage of this.

One of those members is `public` which is a `list` of public methods.

Our class generator definition now looks like

```
MyClass <- R6::R6Class(
  classname = "MyClass",
  public =
    list(
      dataset = NULL
    )
)

MyClass$set(
  which = "public",
  name = "initialize",
  value = source(file = "initialize.R")$value,
  overwrite = FALSE
)
```

which is much cleaner, especially so as the number of methods—and the lengths of methods (in lines)— grows.

Note: `source()`, if assigned, returns a list with `value` and `visible`, therefore set `value = source()$value`.

6.2 Debug

6.2.1 `cannot add bindings to a locked environment`

Member has not been added to `public` or `private`. E.g.

```
MyClass <- R6::R6Class(
  classname = "MyClass",
  public =
    list(
      read_csv = function(path_to_csv) {
        self$dataset <-
          read.csv(
            file = path_to_csv,
            stringsAsFactors = FALSE
          )
      }
    )
)
```

In the above class definition, **`self$dataset`** isn't a member. To fix

```
MyClass <- R6::R6Class(
  classname = "MyClass",
  public =
    list(
      dataset = NULL,
      read_csv = function(path_to_csv) {
        self$dataset <-
          read.csv(
            file = path_to_csv,
            stringsAsFactors = FALSE
          )
      }
    )
)
```

6.3 Terminology

members methods, fields
methods functions that are assigned to objects after instantiated by a class generator
fields data or data storage assigned to objects after instantiated by a class generator

6.4 Notes

class methods vs member functions class methods have access to self; member functions do not.

self$ private$ super$

R6 does not indicate if a method is syntactically incorrect when using $set(). In fact, it compiles the generator without those class methods.

7 Regular expressions (regex)

Brackets

logic, metacharacters without quantifiers, metacharacters with quantifiers keywords for common: -: range

this	that
:digit	digits
:lower	lowercase
:upper	uppercase
:alpha	alphabetic
:alnum	alphanumerics
:punct	punctuation
:blank	space or tab
:space	space, tab, newline, vertical tab, form feed, return

Logic

^: negate brackets uppercase metacharacters negate lowercase versions (`\D` not a digit)

Metacharacters are predefined `[]`s

metacharacter	characters
`\w`	A-z, 0-9
`\s`	space
`\h`	horizontal space
`\v`	vertical space
`\n`	newline
`\r`	carriage return
`\t`	tab
`\b`	word boundary

Quantifiers always follow `[]`s or metacharacters

quantifier	searches for
`*`	0 or more
`+`	1 or more
`{x}`	exactly
`{m, }`	at least
`{, n}`	at most
`{m, n}`	minimum, maximum
`?`	optional

Equivalencies

`\d`, `:digit:`

`+`, `*`, `?` are greedy by default make lazy using a trailing `?` (e.g. `+?`)

Full list of metacharacters:

- `(`
- `)`
- `[`
- `]`
- `{`
- `}`
- `^`
- `$`
- `.`
- `\`
- `?`
- `*`
- `+`
- `|`

`stringr`

locate: start and end extract: match:

7.1 Terminology

word boundary :

non-word boundary :

Capture groups () are for matching ^: string beginning $: string end

7.2 Lookarounds

<: lookbehind !: negative

positive :

negative :

lookahead :

lookbehind :

Let

`x` be a regex of stuff you want `y` be a regex of stuff you don't want

`x(?=y).` This is called *lookahead* `(?<=y)x` This is called *lookbehind*

`(?!query) (?<!query)`

8 {rlang} quoting functions

arguments	developer	user
one	`expr()`	`enexpr()`
many	`exprs()`	`enexprs()`

`{rlang}` quoting functions

- have a more consistent naming scheme than `{base}` quoting functions
- allow for unquoting (thus are also quasiquoting functions), whereas `{base}` quoting functions don't
  - `bquote()` is an exception?
-

Inverses of quoting are

- unquoting (inside `expr()`, e.g. `expr(!!x)` is equivalent to `x`)
- evaluation (outside `expr()`, e.g. `eval(expr(x))` is equivalent to `x`)

8.1 Defusion

8.1.1 Defusing operators (i.e. *defusors*)

Defusing is a synonym for *quoting*. Defusing prevents evaluation of R code. Think of defused expressions as blueprints. The inverse of defusion is *resumption*, i.e. the inverse of defuse is *resume*.

`expr()` and `enquo()` are *defusing operators*, which ensure their arguments (R code) are not evaluated. The former is for the developer, and the latter is for the user.

`name` objects (i.e. symbols) which point to an object in an environment are *environment variables*. `name` objects that refer to a column in a `data.frame` are *data variables*.

`expr()` can create `call` objects (e.g. `expr(foo)`) or `name` objects (e.g. `expr(mean(foo, na.rm = TRUE))`).

8.1.2 Defused expressions

Defused expressions are

- calls (`call` objects), e.g. `mean(c(99, 82, 16))`, ``+`(2, 3)`` (same as `2 + 3`)
- symbols (`name` objects, i.e. object names)

8.2 Forcing operators (i.e. *forcers*)

`!!` and `!!!` are *forcing operators*, which force evaluation inside of a defused expression. Note: It is sometimes necessary to wrap forcing operators in parentheses, e.g. `(!!this)` and `(!!!that)`.

8.3 Unquoting

`!!` unquotes a single argument, which can be a

- `call` object, e.g. `x <- expr(-1)`
- `name` object (i.e. a symbol), e.g. `a <- sym("y")`
- `numeric` object (i.e. a constant), e.g. `b <- 1`

`!!` can be unquote arguments in a function call, or can be used within a function definition:

```r
my_sample <- rnorm(n = 30)
my_expr <- expr(mean(x = !!my_sample))
eval(expr = my_expr)
```

`!!` can also unquote a function. E.g., let's build `var(x, y)`:

```r
f <- expr(var) # quote (`f` is a `name` object)
expr(!!f) # unquote
expr((!!f)(x, y)) # unquote, requote
```

Note: `expr(!!f(x, y))` unquotes the result of `f(x, y)`. We only want to unquote `f`, which is of class `name`.

Example of unquoting a call:

```r
f <- expr(base::list.files) # `f` is an `call` object.
path <- "path/to/files"
pattern <- "\\.csv"
```

```
expr((!!f)(path = path, pattern = pattern))

expr((!!f)(path = !!path, pattern = !!pattern))
```

Also,

```
call2(.fn = f, expr(path), expr(pattern))
call2(.fn = f, expr(!!path), expr(!!pattern))
```

`=` is not allowed in `expr()`. E.g., `expr(path = !!path)` is *not* valid, so

```
call2(.fn = f, expr(path = !!path), expr(pattern = !!pattern))
```

is also not valid. To explicitly set arguments, use ‘`` =` ``()`:

```
call2(.fn = f, expr(`=`(path, !!path)), expr(`=`(pattern, !!pattern)))
```

8.4 Advanced (from `{rlang}` documentation)

`qq_show()` can be used to experiment with

8.5 Analogs

`{base}`	`{rlang}`	this
`quote()`	`expr()`	equivalent
`substitute()`	`enexpr()`	approximate
`alist()`	`exprs()`	equivalent
`as.list(substitute(...()))`	`enexprs()`	equivalent

arguments	developer	user
one	`quote()`	`substitute()`
many	`alist()`	`as.list(substitute(...()))`

8.6 Quasiquototing functions (`{rlang}`)

8.6.1

`!!` is unquote (and for fun can be read as bang-bang)

If a function

- evaluates its arguments, we must do the quoting
- quotes its arguments, we must do the unquoting (with `!!`)

Quoted arguments must be captured by the function and processed.

Nonstandard evaluation

8.6.2 Quasiquotation

Quasiquotation is

1. quotation
2. unquoting
3. non-quoting
4. …

8.6.3 Quoting functions

`expr()`

- ignores whitespace
- is not useful inside of a function
    - use `enexpr()`, which is an *enriched* `expr()`
    - use `enexprs()` for capturing `...`

`exprs(x = x^2, y = y^3, z = z^4)` is equivalent to

```r
list(
  expr(x^2),
  expr(y^3),
  expr(z^4)
)
```

Use

- `enexpr()` and `enexprs()` to capture user input
- `expr()` and `exprs()` to capture own input

Use

`ensym()` and `ensyms()`

8.6.3.1 Table

function	context
`exprs()`	interactive

8.6.4

interactive context user-supplied, variable
non-interactive context developer-supplied, fixed
quotation capturing an expression without evaluating it

8.7 Symbols vs. expressions

symbol :

expression :

9 `{tidyeval}`

9.1 Terminology

pronoun `.data`

quasiquotation

quosures data structure storing both expression and environment

tidyeval underlying toolkit

quosure A special type of formula: A one-sided formula.

`!!`

`quo()` Equivalent to `quote()`?

`enquo()` Equivalent to `substitute()`?

promise

`quo_name()` :

Enclosure When an object keeps track of its environment. (Try `typeof(mean)`)

See `rlang::.data`.

How to evaluate input, rather than quote it? Using a function that captures the expression and environment.

`quote()` and `~` don't work very well, so `quo()` was created.

`quo()` works like `"`: it quotes its input rather than evaluating it.

Use `!!` to

`enquo()`

The definition of `quo()`:

```
quo <- function(expr) {
  enquo(expr)
}
```

```
iris
```

```
column <- "Species"

my_quosure <- quo(column)

blah <- select(.data = iris, !!my_quosure)

blah
```

Does it just boil down to using `enquo()` when using `dplyr` functions inside of my own functions?

Want to `return()` custom strings? Use `quo_name()`. Also, use `:=`.

Multiple arguments?

`...` as a formal argument `quos()` to capture arguments `!!!` to splice

```
x <- c(1:10)

args <- list(na.rm = TRUE, trim = 0.25)

quo(mean(x, !!!args))

# or

args <- list(quo(x), na.rm = TRUE, trim = 0.25)

quo(mean(!!!args))
```

Quoting is capturing. In R, quoting is done via

`~` `quote()`

```
class(~junk)
class(quote(junk))
```

`formula`s capture their code and its execution environment.

get_expr() get_env() eval_tidy()

9.2 Quasiquotation

Quasiquotation LISP

Unquoting:

- basic
- unquote splicing
- unquoting names

Capture `letters` as an expression:

```
quo(toupper(letters))
```

Capture what `letters` would typically return:

```
quo(toupper(!!letters))
```

```
thing <- quo(letters)

quo(toupper(!!thing))
```

The ^ signifies what?

```
quo(list(!!!letters))
```

If you want to unquote on the left-hand side, i.e. set variable names, use `:=`.

```
# From "Tidy evaluation in 5 mins"

my_scatterplot <- function(df, xvar, yvar) {

  xvar <- enexpr(xvar)
  yvar <- enexpr(yvar)

  ggplot(df, aes(!!xvar, !!yvar)) +
    geom_point()

}

source("path/to/my_scatterplot.R")
```

```
my_scatterplot <- function(df, xvar, yvar) {

  browser()
  # > xvar
  # Error: object 'cyl' not found
  # > yvar
  # Error: object 'qsec' not found

  # Promise evaluation?

  xvar <- enexpr(xvar)
  yvar <- enexpr(yvar)

  # > xvar
  # cyl
  # > yvar
  # qsec

  browser()

  ggplot(df, aes(!!xvar, !!yvar)) +
    geom_point()

}

my_scatterplot(
  df = mtcars,
  xvar = cyl,
  yvar = qsec
)
```

10 Under the hood

10.1 Libraries

In R, a library is a directory whose subdirectories are named after and include the contents of R packages.

The types of libraries are

- Common (library is created during install; houses "standard" and "recommended" packages that are installed along with R [see `installed.packages(priority = "high")`])
- User (librar is created during install; houses additional packages installed by the user)
- Site (not created by default)

Site

Sys.getenv() will display all environment variables. It seems to be a superset of what is returned by `set` in Command Prompt, including R-specific variables.

`Sys.getenv("R_HOME") Sys.getenv("R_LIBS_USER") Sys.getenv("R_LIBS_SITE")`

Additional libraries can be added using `.libPaths()`. The order of `.libPaths()` indicates

It is suggested to use a combination of common library and user libraries.

User-specific libraries: Set r-libs-user in /etc/rstudio/rsession.conf

User libraries are associated with major.minor, not major.minor.patch

10.2 References

https://cran.r-project.org/doc/manuals/r-release/R-admin.html#Managing-libraries https://support.rstudio.co
us/articles/215733837-Managing-libraries-for-RStudio-Server

11 Configuration

- `Rprofile.site` (located in installation directory)
- `.Rprofile` (located in home folder [“~/”])

11.1 Dotfiles

Dotfiles are for customization.

Must end with newline or last line will not be ran.

`.Renviron` is for

- sensitive information (e.g. API keys)
- R-specific environment variables

but doesn’t contain R code.

`.Rprofile`

- is for and contains R code
- lives in `R_PROFILE_USER` (`/` by default)

11.2 Libraries

11.2.1 Installing packages

All installation methods use R CMD install, which can do so from source, bundle, or binary packages.

Ways to install packages:

- `install.packages()`
- `devtools::install()`
- `devtools::build()`
- `devtools::install_github()`

11.2.2 Libraries

A library is a directory containing installed packages.

`.libPaths()` shows active libraries. (“Active” meaning that R knows where to find them.)

11.2.3 Where packages are installed

11.2.4 How to install R packages

11.2.5 Via `install.packages()`

11.2.5.1 Installing from a zip

11.2.6 Via `devtools::install.github()`

11.3 Problems with installing R packages

11.3.1 Unable to move temporary installation

When attempting to install a package via `install.packages()`, you may get the warning `unable to move temporary installation 'C:\path\to\temporary\installation' to 'C:\path\to\permanent\installation'`. This problem is due to antivirus and everything moving too fast.

An actual instance looks like this:

```
package 'rgdal' successfully unpacked and MD5 sums checked.

Warning in install.packages :
  unable to move temporary installation
  'path\to\R\win-library\3.4\file26a846732c06\rgdal'
  to
  'path\to\R\win-library\3.4\rgdal'

The downloaded binary packages are in
  path\to\AppData\Local\Temp\1\RtmpmklLST\downloaded_packages
```

11.3.1.1 Solution 1 (preferred)

This solution worked for me:

1. Submit `debug(utils:::unpackPkgZip)`
2. Submit `install.packages("rgdal", dependencies = TRUE)`
3. Step through until complete.

11.3.1.2 Solution 2 (not attempted, but would attempt)

R also prints the location of the

I didn't try this solution, but I would feel comforable doing so. Locate the binary package location (`C:/path/to/downloaded_packages`) as reported by R.

```
zipfile <-
  list.files(
    path = "C:/path/to/downloaded_packages",
    full.names = TRUE
  )

exdir <- libPaths()[1]

for(i in 1:length(path_to_binary)) {

  unzip(
    zipfile = zipfile[i],
    exdir = exdir
  )

}
```

11.3.1.3 Solution 3 (not attempted, but would attempt)

This solution also involves using the zip located in `C:/path/to/downloaded_packages`:

```
pkgs <- "C:/path/to/downloaded_packages/package.zip"

lib <- libPaths()[1]

install.packages(
  pkgs = pkgs,
  repos = NULL,
  type = "win.binary",
  lib = lib
)
```

11.3.1.4 Solution 4 (not preferred)

I initially attempted this solution, but a `Sys.sleep` time of `2.5` didn't work for me:

1. Submit `trace(utils:::unpackPkgZip, edit = TRUE)`
2. Change `Sys.sleep(0.5)` to `Sys.sleep(2.5)`

In addition, this is not a *comfortable* solution, in my opinion. (`utils:::unpackPkgZip()` does get reset after quitting or terminating the R session.)

11.3.1.5 Solution 5 (nope)

The worst solution seems to be changing the read and write access to folders, and this is not recommended as they are probably set as such for a good reason!

References