R Essentials for advanced computing

Jonathan Chipman, Ph.D.

Introduction

Collaborative learning

Google Doc for sharing code/notes

Preparation

Read/Watch: Selections from R Programming for Data Science

- Light read: Chapters 1-3
- Careful read: Chapters 4, 9-10, and 13
- Video: Subsetting objects
- Video: Vectorized Operations

Read: Selections from The Art of R Programming

- Contents through Chapter 1 (Light read for familiarity)
- Chapter 2: Vectors (Careful read)
- Chapter 3, sections 5 - end: Matrixes and Arrays
- Chapter 4, sections 1 - 3 and 5: Lists
- Chapter 7, sections 3-4

Note: Big-picture principles in chapter 2, on vectors, have relevance to matrixes, lists, and data frames.

Aside: A reference for good coding practices

Warm-up problem

Install the package `beepr` and run the command `beepr::beep()`. `beepr::beep(k)` can play k=1-11 sounds. (See `?beep` for a list of sounds).

A. Write loops and functions.

1. Write a loop to listen to each sound. Use `Sys.sleep()` to pause 2 seconds between each call to `beep`.
2. Modify the loop to pause a random duration of time. You can set your own parameters for 'what a random duration of time' means.

B. How can the `beepr::beep()` function be helpful? What does this say about R being a scripting language?

C. After calling `library(beepr)`, the beep function can be directly called as `beep()`. Why can it be helpful to call `beepr::beep()`?

D. (If time allows) Create a function that takes two inputs: a numeric vector and `sound`. Write a loop that one-at-a-time calculates the cumulative sum each element of the vector (don't use the `cumsum` function). Play a beep at the end of calculation using the `sound` input. The output should be a two-column matrix with the original vector (column 1) and the cumsum (column 2). Check you answer using the `cumsum` function.

Note: Question D is designed to practice working with loops and building up the output result. One-at-a-time calculations discouraged whenever avoidable. In practice, it would be better to use the `cumsum` function.

This week's lesson

Focus on foundations

Many of the this week's topics will be familiar, and it may be tempting to gloss over. However, there are important foundational concepts which can strengthen understanding and efficient coding.

Key concepts

In addition to general concepts, the below strategies improve efficient, reproducible code:

1. Use matrices rather than data.frames whenever possible; they use less memory
2. Use names for indexing and filtering; it is transparent and less error-prone
3. Pre-initialize data-structures to be filled rather than saving over existing objects
4. Consider if there are ways to reduce unnecesary calculations

R and RStudio

What is R?

R is an object oriented-, functional-, and scripting-programming language.

Object-oriented programming language

- R Manual: Objects
- Everything in R is an object.
- Data are stored in objects (vectors, matrixes, lists, data.frames) and manipulated using objects (functions).

- Objects:
  - Are assigned a value via `<-` (preferred), `=`, or `->`.
  - Have basic, intrinsic properties (aka `attributes`): `mode` (data type) and `length`
  - May have additional `attributes` such as (list from link)
    - class (a character vector with the classes that an object inherits from).
    - comment
    - dim (which is used to implement arrays)
    - dimnames
    - names (to label the elements of a vector or a list).
    - row.names
    - levels (for factors)
  - Have a `class` which may behave differently for generic functions (such as `plot` and `summary`) … we'll discuss classes later.

The attributes of an object can be seen through `str()` and `attributes()`:

```
data("HairEyeColor")
HairEyeColor
```

```
, , Sex = Male

       Eye
Hair    Brown Blue Hazel Green
  Black    32   11    10     3
  Brown    53   50    25    15
  Red      10   10     7     7
  Blond     3   30     5     8
```

```
, , Sex = Female

       Eye
Hair    Brown Blue Hazel Green
  Black    36    9     5     2
  Brown    66   34    29    14
  Red      16    7     7     7
  Blond     4   64     5     8
```

```
str(HairEyeColor)
```

```
 'table' num [1:4, 1:4, 1:2] 32 53 10 3 11 50 10 30 10 25 ...
 - attr(*, "dimnames")=List of 3
  ..$ Hair: chr [1:4] "Black" "Brown" "Red" "Blond"
  ..$ Eye : chr [1:4] "Brown" "Blue" "Hazel" "Green"
  ..$ Sex : chr [1:2] "Male" "Female"
```

Functional programming language

- Perform operations on object(s) (ex: `sum`, `'+'`, `rnorm`)
- Functions come pre-intalled (base), installed, and custom-defined
- Using functions is a major theme of good R programming. Avoid explicit iteration (loops and copy-paste) as much as possible. [Matloff (pg xxii)]:
  - Clearer, more compact code
  - Potentially must faster execution speed
  - Less debugging, because the code is simpler
  - Easier transition to parallel programming
- R Manual: Writing-your-own-functions

```
# Example of base function (with R installation)
1+1

# Example of installed function from a package
install.packages("beepr")
beepr::beep(0)

# General structure of custom function
< name of your function > <- function(
```

```
    < argument 1>,
    < argument 2> = < default value >,
    ...,
    < argument n>
) {

  < some R code here >

  return(< some result >)

}
```

```
# Example of custom function
fun <- function(v1,v2) {v1+v2}
fun(1,2)
```

- Scripting language
  - R Manual: Scripting-with-R
  - A script is run top-to-bottom. It is reproducible and transparent.
  - Can be run in 'interactive' (ex: RStudio) and 'batch' modes (ex: command-line call)
  - A reference for good coding practices

What is RStudio?

- An Integrated Development Environment (IDE) to organize and facilitate common tasks coding in R

Data modes and structures

Modes

Six primitive modes (data types) of R:

2 Modes less-commonly (for me never) created in practice:

- Raw (raw byte) `x <- raw(2)`
- Complex `x <- 0i`

1 Mode moderately created (often indirectly through R) in practice:

- Integer `x <- 1L`

3 Modes commonly created in practice:

- Logical (TRUE/FALSE) `x <- TRUE`
- Numeric (real number) `x <- 0`
- Character / String `x <- "hello world`

Structures

Data are stored in any of the following structures (starting from most primitive):

- vector: A set of elements having the same mode (data type).
- matrix/array: A vector with dimensions.
  - Matrices have rows and columns.
  - Arrays can have any number of dimensions.
- list: A set of elements which may have different modes / structures
- data.frames: A list shaped like a matrix

```
# Numeric vector
x <- 1:10
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
# Matrix of numerics with rownames 1, 2, and 3
rbind("1"=x,"2"=x,"3"=x)
```

```
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
1    1    2    3    4    5    6    7    8    9    10
2    1    2    3    4    5    6    7    8    9    10
3    1    2    3    4    5    6    7    8    9    10
```

```
# List with element names 1, 2, and 3
x <- list("1"=1, "2"=2, "3"="A")
x
```

```
$`1`
[1] 1

$`2`
[1] 2

$`3`
[1] "A"
```

```
# data.frame
as.data.frame(x)
```

```
  X1 X2 X3
1  1  2  A
```

Back to modes … The 6 primitive modes are also called 'atomic' modes. This means that when data are stored together in a vector, they must be of the same mode.

When modes are mixed (ex: `x <- c("A",0,1L,TRUE)`), R will force all elements to be of the same mode.

Question: What is your guess for which modes receive greatest priority?

Missing values

R has different types of missing values:

- `NA`: no information, has length 1,
- `NULL`: which has length 0,
- `Inf`: Infinite, and
- `NaN`: Not a Number
- These have companion functions `is.na()`, `is.null`, `is.infinite` (or `is.finite()`, which covers NA, Inf, and NaN), and `is.nan`.

Questions 1

1. What is the mode of the following vector `myVector <- c(NA, NaN, Inf)`? (First try to answer without coding, then check using the `mode()` function in R)
2. The `c()` function can be used with other vectors, for example

```
myNumericVector <- c(1, 2, 3)
myStringVector  <- c("hello", "world")
```

   What is the mode of the vector `c(myStringVector, myStringVector)`?
3. What do each one of the functions `is.na`, `is.null`, `is.finite`, `is.infinite`, `is.nan` return on the vector `myVector`?
4. What are the attributes of the following object `myMat <- matrix(runif(12), ncol=4)`? What are the attributes of `myNumericVector` and how can you make sense of the attributes?

Vectors and matrixes

Why to use vectors and matrices: they use less memory than lists and data.frame.

Efficient programming in R matrix vs data table vs data frame

Get into a habit of using vectors and matrices / arrays as much as possible.

Creating vectors

- To initialize an empty vector, use `vector`, `rep(NA,<length>)`, `numeric(<length>)`, or `character(<length>)`. (JC) I commonly use `rep(NA,<length>)`.
  - A question to keep in back of mind (we will revisit when talking about loops) … Why would you want to initialize an empty vector?

```
# vector(< mode >, < length >)
vector("character",2)
```

```
[1] "" ""
```

```
vector("numeric",2)
```

```
[1] 0 0
```

```r
vector("logical",2)
```

```
[1] FALSE FALSE
```

```r
rep(NA,2)
```

```
[1] NA NA
```

```r
numeric(2)
```

```
[1] 0 0
```

```r
character(2)
```

```
[1] "" ""
```

- Other common methods to create numeric vectors include **combine** function, `c()`, `:`, `seq`, and `rep`:

```r
# Combine 3 numeric vectors each with length 1
c(1,2,3,4)
> [1] 1 2 3 4

# Vector of sequential numerics
x <- 1:10
x
>  [1]  1  2  3  4  5  6  7  8  9 10

# Vector of sequential numerics from 1 to length(x)
seq(x)
>  [1]  1  2  3  4  5  6  7  8  9 10

# Vector of sequential numerics from 1 to 10 by 2
seq(1,10,by=2)
> [1] 1 3 5 7 9

# Vector of sequential numerics from 1 to 10 divided equally into 3 elements
seq(1,10,length.out=3)
> [1]  1.0  5.5 10.0
```

```r
# Vector of repeated numerics
rep(1:2,each=5)
>  [1] 1 1 1 1 1 2 2 2 2 2

# Vector of repeated numerics
rep(1:2,times=5)
>  [1] 1 2 1 2 1 2 1 2 1 2
```

Creating matrices

- Three common ways to create matrices: `matrix`, `cbind`, `rbind`.

```r
x <- 1:10
rbind(x,x,x)
```

```
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
x    1    2    3    4    5    6    7    8    9    10
x    1    2    3    4    5    6    7    8    9    10
x    1    2    3    4    5    6    7    8    9    10
```

```r
cbind(x,x,x)
```

```
       x  x  x
 [1,]  1  1  1
 [2,]  2  2  2
 [3,]  3  3  3
 [4,]  4  4  4
 [5,]  5  5  5
 [6,]  6  6  6
 [7,]  7  7  7
 [8,]  8  8  8
 [9,]  9  9  9
[10,] 10 10 10
```

```r
matrix(x,nrow=2)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Naming and indexing

Elements of vectors and matrices can be extracted through `[<position(s)>]` or `[<name(s)>]`.

```
# Vector[k] pulls out the element(s) indexed by k
x <- 1:10
x[3]
```

```
[1] 3
```

```
x[c(5:3)]
```

```
[1] 5 4 3
```

```
names(x) <- 2011:2020
x
```

```
2011 2012 2013 2014 2015 2016 2017 2018 2019 2020
   1    2    3    4    5    6    7    8    9   10
```

```
x[c("2015","2017")]
```

```
2015 2017
   5    7
```

```
# Matrix[r,c] pulls out the element(s) indexed by row(s) r and columns (c)
x <- matrix(1:10,nrow=2)
x
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```
x[1,c(2,5)]
```

```
[1] 3 9
```

```r
rownames(x) <- 2010:2011
colnames(x) <- c("SLC","Murray","Bountiful","Milcreek","Sandy")
x
```

```
     SLC Murray Bountiful Milcreek Sandy
2010   1      3         5        7     9
2011   2      4         6        8    10
```

```r
x[,c("SLC","Milcreek")]
```

```
     SLC Milcreek
2010   1        7
2011   2        8
```

```r
# Beware of dimension reduction
dim(x[,c("SLC","Milcreek")])
```

```
[1] 2 2
```

```r
dim(x["2010",c("SLC","Milcreek")])
```

```
NULL
```

In the last example, why is the dimension `NULL`? (Hint, what is the class of the last two examples?)

Questions 2

1. What is the number of the alphabet for each letter of your name? Use a vector with names (try the 'LETTERS' object). For example, if the letters to the name JONATHAN were mapped to integers, the result would be: 10, 15, 14, 1, 20, 8, 1, 14.
2. Why is it important extract elements through naming conventions?

Lists

Why to use lists?

- Lists are useful for returning output from functions. For example, the output of `lm` is a list. (Aside, it is also of the `class` lm which has a specific behavior when calling "generic" functions such as `print` and `summary`).

```
f <- lm(1:10~1)
names(f)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "call"          "terms"         "model"
```

```
# Two equivalent plot statements
# plot(f,ask = FALSE)
# plot.lm(f,ask=FALSE)

# Two equivalent summary statements
# summary(f)
# summary.lm(f)
```

Creating lists

```
x <- list(1:2,"A",NULL,c(TRUE,FALSE),list(1:10,"B"))
x
```

```
[[1]]
[1] 1 2

[[2]]
[1] "A"

[[3]]
NULL

[[4]]
[1]  TRUE FALSE
```

```
[[5]]
[[5]][[1]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[5]][[2]]
[1] "B"
```

```
x <- list("2010"=1,"2011"="A","2012"=NULL,"2013"=TRUE)
x
```

```
$`2010`
[1] 1

$`2011`
[1] "A"

$`2012`
NULL

$`2013`
[1] TRUE
```

Naming and indexing

Names can be set as in the example above, or through the `names` argument.

```
x <- list(1,2,3)
names(x) <- c("A","B","C")
x
```

```
$A
[1] 1

$B
[1] 2

$C
[1] 3
```

List elements can be extracted using `[]`, `[[]]`, or `$`. See Subsetting Lists.

```
x <- list("2010"=1:2,
          "2011"="A",
          "2012"=NULL,
          "2013"=c(TRUE,FALSE),
          "2014"=list("H1"=1:10,"H2"="B"))
x
```

```
$`2010`
[1] 1 2

$`2011`
[1] "A"

$`2012`
NULL

$`2013`
[1]  TRUE FALSE

$`2014`
$`2014`$H1
 [1]  1  2  3  4  5  6  7  8  9 10

$`2014`$H2
[1] "B"
```

```
# Extract multiple elements
x[as.character(2010:2012)]
```

```
$`2010`
[1] 1 2

$`2011`
[1] "A"

$`2012`
NULL
```

```r
# Extract single elements
x[["2010"]]
```

```
[1] 1 2
```

```r
# Another option to extract by name
x$"2010"
```

```
[1] 1 2
```

data.frames

Why to use data.frames?

- data.frames are useful in data analysis.

- However, a data.frame of all one mode uses more memory than a matrix. When performing heavy computations, avoid data.frames unless needed.
- A helpful data.frame comes from `expand.grid`. How can this be helpful in simulations?

```r
expand.grid("Param 1"=1:2,"Param 2"=letters[1:3])
```

```
  Param 1 Param 2
1       1       a
2       2       a
3       1       b
4       2       b
5       1       c
6       2       c
```

data.frames can be named and indexed similarly as above for lists. (A data.frame is a list.)

Naming and indexing

Rule of thumb: When you want to extract information from a data structure, use names (rather than position indexes).

- Transparent in code and output
- Easier to read
- Reproducible

Borrowing from Blog on R code best practices (one user's opinion): There are 5 naming conventions to choose from:

- alllowercase: e.g. adjustcolor
- period.separated: e.g. plot.new
- underscore_separated: e.g. numeric_version
- lowerCamelCase: e.g. addTaskCallback
- UpperCamelCase: e.g. SignatureMethod

Strive for names that are concise and meaningful

Naming conventions are personal preference. My (JC) inclination is the following:

- I use lowerCamelCase for all objects and files. (Python uses the period to access functions and properties of objects).
- Avoid naming an object that would overwrite an existing object. (Example, `c <- 0` overwrites the `c()` function).
- Clarity is better than brevity. (Example, `s` in `s <- 0` could stand for a method name, simulation, etc.).

Vectorizing

R Simultaneously performs the same operation across vector elements.

- Behind the scenes R calls C code to do one-at-a-time calculation, but this is faster than doing one-at-a-time calculation in R.

Element-wise addition in one call

```
x1 <- 1:10
x2 <- 101:110
x1 + x2
```

```
 [1] 102 104 106 108 110 112 114 116 118 120
```

Element-wise logic assessments in one call

```
x1 > 5
```

```
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

The above is an example of R 're-cycling' 5 to have the same length as x1. It is a strength and a caution.

```
x <- 1:10
y <- 2:4
cbind(x,y)
```

```
Warning in cbind(x, y): number of rows of result is not a multiple of vector
length (arg 2)
```

```
      x y
 [1,]  1 2
 [2,]  2 3
 [3,]  3 4
 [4,]  4 2
 [5,]  5 3
 [6,]  6 4
 [7,]  7 2
 [8,]  8 3
 [9,]  9 4
[10,] 10 2
```

```
x < y
```

```
Warning in x < y: longer object length is not a multiple of shorter object
length
```

```
 [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Key point: Loops (to be discussed further) carry out one-at-a-time operations. Usually, a vectorized alternative to loops is to use *apply functions with loops. This will be discussed in more later in the class.

Filtering vectors

Elements of a vector can be filtered through:

- The vector position (aka index),
- The element's name, or
- Boolean logic (TRUE/FALSE)
  - `==` Test for equality
  - `>`, `>=` Test for greater than and greater than or equal to
  - `<`, `<=` Test for less than and less or equal to

Example, 10 patients were randomly assigned to control (0) or treatment (1). Suppose `tx` is the treatment assignment.

```
tx <- rep(0:1,times=5)
names(tx) <- 1:10

tx[tx==1]
```

```
 2  4  6  8 10
 1  1  1  1  1
```

```
tx[c(1,3,7)]
```

```
1 3 7
0 0 0
```

```
tx["3"]
```

```
3
0
```

- Boolean logic can be used with
  - `which`: which vector positions meet a testing criteria
  - `any` and `all`: Boolean test (yes/no) for whether any or all elements meet criteria

```
which(tx==1)
```

```
 2  4  6  8 10
 2  4  6  8 10
```

```
tx[which(tx==1)]
```

```
 2  4  6  8 10
 1  1  1  1  1
```

```
any(tx==1)
```

```
[1] TRUE
```

```
all(tx==1)
```

```
[1] FALSE
```

Questions 3

Suppose `x` is the number of vacations in the years 2010 - 2019

```
set.seed(1)
x <- sample.int(n=7,size=10,replace=TRUE)
names(x) <- 2010:2019
```

1. How many vacations were taken on the first and ninth year?
2. How many total vacations were taken on odd years? Use element names for solution.
3. How many total vacations were taken on even years? Use the `seq` function for solution.
4. Why would you want to use element names whenever possible?

–> –>

Control Statements

"R is a block-structured language … delineated by braces, though braces are optional if the block consists of just a single statement. Statements are separated by newline characters or, optionally, by semicolons." (Matlooff, page 139)

If-then statements

Conditional logic evaluates

```
x <- "yes"
if(x=="yes") {
  print("I'll take on the project")
} else {
  print("Sorry, I can't take on the project")
}
> [1] "I'll take on the project"
```

The above `if-else` statement requires a single TRUE/FALSE evaluation. `ifelse` vectorizes conditional logic.

```
x <- seq(1:10)
ifelse(test = x>5, yes = 1, no = 0)
>  [1] 0 0 0 0 0 1 1 1 1 1
```

Loops

Loops iterate operations through a parameter saved in a vector. Possible loops include:

- `for`: iterate through each value/element of a vector
- `while`: continue loop while TRUE until FALSE
- `repeat`: continue loop until a `return` or `break` statement

```
x <- 1:10
for (n in x){
  print(n)
}
> [1] 1
> [1] 2
> [1] 3
> [1] 4
> [1] 5
> [1] 6
> [1] 7
> [1] 8
> [1] 9
> [1] 10
```

```
for(n in 1:length(x)){
  print(x[n])
}
> [1] 1
> [1] 2
> [1] 3
> [1] 4
> [1] 5
> [1] 6
> [1] 7
> [1] 8
> [1] 9
> [1] 10
```

What will the following return:

```
x <- seq(1,10,by=3)
for(i in x){
  print(i)
}
```

```
i <- 1
while (i <= 10){
  i <- i + 4
}
i
> [1] 13

i <- 1
while(TRUE){
i <- i + 4
if(i > 10) break
}
i
> [1] 13

i <- 1
repeat{
  i <- i + 4
  if (i > 10) break
}
```

```
i
> [1] 13
```

The `next` statement allows the loop to stop current iteration and continue to next iteration.

Questions 4

From Jan 1 - Jan 9, 2023, it snowed 6 days atop Snowbasin. Snow days can be represented each day in a vector as 1 (snow) and 0 (no snow).

```
snow <- c(1,1,1,1,0,1,1,0,0)
names(snow) <- 1:9
```

Suppose you are interested in the first day it consecutively snowed three days (i.e. snowed the given day and two previous days). What is this day? Solve using a loop with conditional logic.

Functions

- Repeating the strengths of functions … Using functions is a major theme of good R programming. Avoid explicit iteration (loops and copy-paste) as much as possible. [Matloff (pg xxii)]:
  - Clearer, more compact code
  - Potentially must faster execution speed
  - Less debugging, because the code is simpler
  - Easier transition to parallel programming
- In general terms, R functions are structured as follow:

```
< name of your function > <- function(
    < argument 1>,
    < argument 2> = < default value >,
    ...,
    < argument n>
) {

  < some R code here >

  return(< some result >)

}
```

- For example, if we want to create a function to run the "unfair coin experiment" we could do it in the following way:

```r
# Function definition

# unfairCoin
# n: number of tosses
# p: biased coin (default = 0.7)
unfairCoin <- function(n, p = 0.7) {

  # Sampling from the coin dist
  ans <- sample(c("H", "T"), n, replace = TRUE, prob = c(p, 1-p))

  # Returning
  ans

}

# Testing it
set.seed(1)
tosses <- unfairCoin(20)
table(tosses)
```

```
tosses
 H  T
13  7
```

```r
prop.table(table(tosses))
```

```
tosses
   H    T
0.65 0.35
```

Questions 5

Generalize your code as a function so that for any string of days you can find the first day it consecutively snowed a given number of days. Return `NA` if no day meets this criteria.

Session info

```
devtools::session_info()
```

```
- Session info -------------------------------------------------------------
 setting  value
 version  R version 4.4.1 (2024-06-14)
 os       macOS Sonoma 14.6.1
 system   aarch64, darwin23.4.0
 ui       unknown
 language (EN)
 collate  en_US.UTF-8
 ctype    en_US.UTF-8
 tz       America/Denver
 date     2024-08-19
 pandoc   3.2.1 @ /opt/homebrew/bin/ (via rmarkdown)

- Packages -----------------------------------------------------------------
 package     * version date (UTC) lib source
 cachem        1.1.0   2024-05-16 [1] CRAN (R 4.4.1)
 cli           3.6.3   2024-06-21 [1] CRAN (R 4.4.1)
 devtools      2.4.5   2022-10-11 [1] CRAN (R 4.4.1)
 digest        0.6.36  2024-06-23 [1] CRAN (R 4.4.1)
 ellipsis      0.3.2   2021-04-29 [1] CRAN (R 4.4.1)
 evaluate      0.24.0  2024-06-10 [1] CRAN (R 4.4.1)
 fastmap       1.2.0   2024-05-15 [1] CRAN (R 4.4.1)
 fs            1.6.4   2024-04-25 [1] CRAN (R 4.4.1)
 glue          1.7.0   2024-01-09 [1] CRAN (R 4.4.1)
 htmltools     0.5.8.1 2024-04-04 [1] CRAN (R 4.4.1)
 htmlwidgets   1.6.4   2023-12-06 [1] CRAN (R 4.4.1)
 httpuv        1.6.15  2024-03-26 [1] CRAN (R 4.4.1)
 jsonlite      1.8.8   2023-12-04 [1] CRAN (R 4.4.1)
 knitr         1.47    2024-05-29 [1] CRAN (R 4.4.1)
 later         1.3.2   2023-12-06 [1] CRAN (R 4.4.1)
 lifecycle     1.0.4   2023-11-07 [1] CRAN (R 4.4.1)
 magrittr      2.0.3   2022-03-30 [1] CRAN (R 4.4.1)
 memoise       2.0.1   2021-11-26 [1] CRAN (R 4.4.1)
 mime          0.12    2021-09-28 [1] CRAN (R 4.4.1)
 miniUI        0.1.1.1 2018-05-18 [1] CRAN (R 4.4.1)
 pkgbuild      1.4.4   2024-03-17 [1] CRAN (R 4.4.1)
 pkgload       1.4.0   2024-06-28 [1] CRAN (R 4.4.1)
 profvis       0.3.8   2023-05-02 [1] CRAN (R 4.4.1)
 promises      1.3.0   2024-04-05 [1] CRAN (R 4.4.1)
```

```
 purrr         1.0.2   2023-08-10 [1] CRAN (R 4.4.1)
 R6            2.5.1   2021-08-19 [1] CRAN (R 4.4.1)
 Rcpp          1.0.12  2024-01-09 [1] CRAN (R 4.4.1)
 remotes       2.5.0   2024-03-17 [1] CRAN (R 4.4.1)
 rlang         1.1.4   2024-06-04 [1] CRAN (R 4.4.1)
 rmarkdown     2.27    2024-05-17 [1] CRAN (R 4.4.1)
 sessioninfo   1.2.2   2021-12-06 [1] CRAN (R 4.4.1)
 shiny         1.8.1.1 2024-04-02 [1] CRAN (R 4.4.1)
 stringi       1.8.4   2024-05-06 [1] CRAN (R 4.4.1)
 stringr       1.5.1   2023-11-14 [1] CRAN (R 4.4.1)
 urlchecker    1.0.1   2021-11-30 [1] CRAN (R 4.4.1)
 usethis       2.2.3   2024-02-19 [1] CRAN (R 4.4.1)
 vctrs         0.6.5   2023-12-01 [1] CRAN (R 4.4.1)
 xfun          0.45    2024-06-16 [1] CRAN (R 4.4.1)
 xtable        1.8-4   2019-04-21 [1] CRAN (R 4.4.1)
 yaml          2.3.8   2023-12-11 [1] CRAN (R 4.4.1)

 [1] /opt/homebrew/lib/R/4.4/site-library
 [2] /opt/homebrew/Cellar/r/4.4.1/lib/R/library
```
