Lecture Notes

STAT 33B with Gaston Sanchez

Fall 2023

Contents

1 Wednesday, August 23rd

1.1 Class Syllabus

1.1.1 Lectures

Introduction to Advanced Programming in R (Fall 2023)

Course Information

Instructor: Gaston Sanchez, OH TBD in Evans 309

First homework will be released on September 8. Homeworks will be released every two weeks.

Midterm: October 13

Final: December 12

Labs are worth 10% of grade, homeworks are worth 65% of grade, midterm is worth 6% of grade, and final is worth 19% of grade.

No rounding up grades when calculating. This semester is different due to the significant demand.

Step 1. Install R.

Step 2. Install R Studio. (Google "Install R Studio")

If you run into installation issues, use r.datahub.berkeley.edu.

R Studio Components

History contains all former commands in case something happens to your code. R automatically creates a .Rhistory file which contains these commands.

Environment keeps track of all the objects created in the session. R stores this information in an .RData file. This is why the console will say "Workspace loaded from .RData".

2 Wednesday, August 30th

2.1 Vectors

We can make a vector as such: `v <- c(1, 2, 3)` where `c` is either the 'combine' or 'concatenate' function.

2.2 Types

R has 4 main data types. In order of size that they occupied from least to greatest:

1. **log**ical
    - ex. `z = TRUE`
2. **int**eger
    - ex. `y = 105L`, where the letter 'L' declares this as an integer
3. **dbl** (double, or 'real' numbers)
    - ex. `x = 105.10`

3.5. **complex**

4. **chr** (character)

2.2.1 Finding Types

We can quantify the size of a variable *var* with `object.size(var)` as something like `56 bytes`. Furthermore we can use `typeof` and `mode`, where the latter is sometimes preferred if we do not need the extra granularity, and the former is preferred when you need to be precise.

`typeof(v[i])` where `v[i]` is `NA`, will give you the type of the other values in the vector `v`.

`typeof(NA)` gives `\logical"`.

2.3 Special Values

2.3.1 NULL

`NULL` is represented in all caps and is not a vector. Furthermore, it is a reserved keyword – you will not be able to assign to it (i.e. using it as a variable name).

2.3.2 Missing Values

These are represented by `NA` which stands for 'Not Available'.

2.3.3 Not A Number

You can get `NaN`'s from negative inputs to the `sqrt` and `log` functions.

2.3.4 Inf

You can get `Inf` or `-Inf` with `1/0` or `-1/0` respectively.

2.4 Mixing Data Types

2.4.1 Implicit Coercion

If you combine different types then you combine types with the resulting vector having the type being the most dominant value.

If you add a `NULL` with any other types, the `NULL`'s will just be dropped.

2.4.2 Explicit Coercion

`as.logical(0)` returns `FALSE`.

`as.logical` returns `TRUE` for any other numeric value.

`as.logical` returns `NA` for any value that takes more memory than a *numeric* value type.

Likewise there are: `as.integer(...)`
`as.double(...)`
`as.character(...)`

Similarly we have testing functions `is.logical(...)`
`is.integer(...)`
`is.double(...)`
`is.character(...)`

Coercion Sanity Check:
Q: What type does everything get converted to in `c(TRUE, 1L, 4L, 6)`?
A: `double`

2.5 Vectorization

Doing elementwise operations all at once!

2.6 Recycling

```
> a = c(1, 2)
> b = c(2, 4, 6, 8)
> a + b
[1]  3  6  7 10
```

Note that R ‘recycles’ or ‘repeats’ the elements in the shorter vector to match the longer one.

If the shorter one’s length is a multiple of the longer one’s length then this occurs without issue.

! However if that is not the case, then we run into a Warning – though it will still execute!

3 Wednesday, September 6th

3.1 Vectors

Vectors are fundamental data structures in R. They can be subsetted, reassigned, and indexed both within and outside their bounds.

3.1.1 Numeric Subsetting

Numeric vectors can be used for subsetting in R. Exclusions can be made using both the column operator and parentheses.

3.1.2 Reassignment

To modify a value at a specific position in a vector, use brackets. For example, to change the value at the $x1$ position in vector x, use: `x[x1] = a`.

3.1.3 Out of Bounds Indexing

- Accessing an index outside the vector's bounds gives an `<NA>` message.
- This can also be used to append elements. For a list with four elements, `x[5] = a` adds a as the fifth element.

3.1.4 Logical Subsetting

- Exclude an index by writing the logical vector. For a vector X with four elements, use `X[c(TRUE, TRUE, TRUE, FALSE)]` to omit the fourth value.
- For comparison, $X > 5$ in [2, 3, 6, 5] gives FALSE, FALSE, TRUE, FALSE. Updating X with these omits the 1st, 2nd, and 4th values.

3.1.5 Double Brackets

- `x[[1]]` returns the first value of vector X.
- `X[[1, 2]]` gives an error as it tries to select more than one element.
- `X[[A]]` should return only one element.
- Both `X[[a]]` and `X[a]` return element a. Single brackets are common, but double brackets, which are faster, return only one element and are used for lists.

3.2 Matrices

Matrices in R are two-dimensional vectors that can store data in rows and columns.

3.2.1 Matrix Representation

- Stored in column-major format.
- Represented as: `matrix(data = c(2, 4, 6, 8), nrow = 2, ncol = 2)`.

3.2.2 Matrix Storage

- First argument is `data =`.
- Stored as vectors but can be matrix objects with rows and columns.

3.2.3 Changing Representation

- Use `byrow = TRUE` for row-major representation.
- Display the matrix by inputting its name.

3.2.4 Displaying Matrices

To view a matrix X as the vector R stores it, use `as.vector(X)`.

3.2.5 Style

For clarity, specify both `nrow` and `ncol` when defining a matrix.

4 Wednesday, September 13th

4.1 Atomic Objects

Definition: Atomic Objects means all of the objects' elements have the same type.

Atomic Objects include vectors (1-dimensional), matrices (2-dimensional), and n-dimensional arrays.

4.2 Lists

A vector is a sequence of continuous cells, while a list is a collection of non-continuous cells.

A list y can be represented as `y = list(a, b, c)`, with each element being its own separate vector.

4.2.1 Element Names

You can assign names to each element using the `names()` function. For our list y, we can set `names(y) = c(‘‘x’’, ‘‘y’’, ‘‘z’’)`, sequentially assigning names to each element.
Consider `z = list(TRUE, 3L, ‘‘hi’’)`.

We can also rename (not the values) of each element using the list constructor:
`w = list(a = TRUE, b = 3L, c = ‘‘hi’’)`.

4.2.2 Subsetting (Subindex, Subscript)

1. **Single brackets**: Using single brackets, we can see both the name and the value of the item we're accessing. For example, `y[1]` evaluates to `a [1] TRUE`.
2. **Double brackets**: Using double brackets, we only see the value of the element we're accessing. For instance, `y[[1]]` evaluates to `[1] TRUE`.

4.2.3 List Operations

List operations in R are fairly limited. For instance, we cannot duplicate lists simply by multiplying with integers. Assume we have a list `a = list(‘‘this’’, ‘‘is’’, ‘‘a’’, ‘‘list’’)`. We cannot use `2 * a` to duplicate the list. Nested elements of a list can be accessed using appropriate indexing.

4.3 Data Frames

A data frame is a non-atomic version of a matrix. Data frames are initialized with:

```
dframe = data.frame(
"name" = c('Luke', 'Leia', 'Han'),
"height" = c(177, 160, 190),
"force" = c(TRUE, TRUE, FALSE)
)
```

4.3.1 Properties/Features of Data Frames

When using `class()` on data frames, R will indicate that it's a data frame. However, `typeof()` on a data frame will reveal that it is essentially a list. Individual columns within a data frame can be accessed using the $ notation, e.g., `dframe$height`. Specific elements can also be accessed using 2D indexing. For instance:

- `dframe[1, 1]` gives the element in the first row and first column.
- `dframe[1, 1:3]` provides elements in the first row across the first to third columns.
- `dframe[1:3, 1]` yields the elements in the first to third rows of the first column.

5 Wednesday, September 20th

5.1 Functions

The function `file()` allows you to open files, `url()` allows you to open URLs, and `unz()` allows you to unzip files. Most of the time, you don't need to explicitly call these functions. They are typically executed behind the scenes during the import/export process.

5.2 Imports

Tabular formats include: .csv, .tsv, and .txt. To read these data table formats, we use `read.table()`. Specific subfunctions of `read.table()` include `read.csv()`, `read.csv2`, `read.delim`, and `read.fwf()`. The output of these functions is a `data.frame`.

There are also provisions to import data from "foreign" packages such as SAS, SPSS, Stata, Matlab, etc.

5.2.1 Unstructured Imports

For unstructured data, `readLines()` reads a character vector. The output is a character vector, not a data frame.

For clarity: Functions like `read.table()` and `readLines()` are high-level, while functions they depend on, such as `scan()`, are low-level.

5.2.2 Use of these functions

The `scan()` function can be used as `scan(file = \filename.txt", what = double())`. You can assign the output to objects. The `sep = \,"` argument can be added to distinguish between elements separated by a specific character. For instance, `scan(file = \filename.filetype", what = double(), sep = \,")` would read each value separated by a comma.

5.3 Reading structured imports

To read a table, you can use `read.table(file = filename.filetype)`. The `header` argument can be added to specify if the data has a header. For example, to assign a table to an object named `tbl` without a header and with elements separated by commas, you would use `tbl = read.table(file = filename.filetype, header = FALSE, sep = ‘‘,’’)`.

5.4 Exports

The functions `save()` and `sink()` can be used for exporting data. The `save()` function saves your content into a binary file, which can only be opened with R. With the `file` argument, you can specify the filename. To choose a specific location in the directory, you can prepend the file path to the name, e.g., `save(tbl, file = ‘‘Desktop/filename.filetype’’)`.

The `sink()` function diverts output. Instead of displaying the output in the console, you can divert it to an external file. You can call `sink()` as: `sink(file = ‘‘output.txt’’)`. Any console output will then be written to `output.txt`. To stop the diversion, call `sink()` with no arguments.

5.5 Linear Regression

You can run a linear regression model on structured data using `lm()`. In the expression `reg = lm(mpg` $\sim$ `cyl + disp, data = mtcars)`, `mpg` is the response variable, and `cyl` and `disp` to the right of the   are the explanatory variables. The `data = mtcars` argument specifies that the data comes from the built-in R `mtcars` dataset. The result is stored in the `reg` object.

6 Wednesday, September 27th

6.1 tidyverse

Tidyverse can be loaded as such:

```r
library(tidyverse)
```

Tidyverse is a collection of packages for data science (whatever that means). There are 8 core packages included:

1. dplyr
2. ggplot2
3. readr
4. forcats
5. stringr
6. tibble
7. tidyr
8. purrr

These "tidy" packages *build* on top of base R objects but with its own base object being the `tibble`.

6.2 tibble

The `tibble` is the modern reimagining of the `data.frame`, you can think of them as mostly the same – though philosophically the `tibble` tries to be more robust and consistent.

6.2.1 Recycling

R dataframes automatically recycle all the time. `tibble` does not recycle at all and will error instead, except in the case where you say `tib$myCol = 2` which will add a column of all 2's.

6.2.2 Column name Autocomplete

R dataframes automatically figure out what you mean when you say `dat$gen` to refer to a column called "gender". `tibble` does not do this and will error unless you use the full name: `tib$gender`.

6.3 dplyr

6.3.1 slice

Select a given row:

1. In base R we say statements like: `dat[1, ]` or `dat[1:5, ]` or etc or etc. But in `tibble` we use *functions* instead.
   The `tibble` equivalents are: `slice(dat, 1)` or `slice(dat, 1:5)` or `slice(dat, c(2, 4))` or `slice(dat, -1)` to exclude the first row.

6.3.2 select

Select a given column:

1. In base R we say statements like: `dat[, 1]` or `dat[, 1:5]` or etc or etc. But in `tibble` we use *functions* instead.
   The `tibble` equivalents are: `select(dat, 1)` or `select(dat, 1:5)` or `select(dat, c(2, 4))` or `select(dat, -1)` to exclude the first row.

6.3.3 filter

This allows you to filter the content of your table based on some condition:

1. In base R we say statements like: `dat[gender == ‘‘female’’,]`. But in `tibble` we use *functions* instead.
   The `tibble` equivalents are: `filter(dat, gender == ‘‘female’’)`.

6.3.4 mutate

This `tibble` command does not modify your dataframe in-place – it adds a new column instead:

1. In base R we say statements like: `dat$height = dat$height * 100` to trasnform an existing column. But in `tibble` we use *functions* instead.
   The `tibble` equivalents are: `mutate(dat, height = height * 100)`.

Note that `mutate(dat, height * 100)` would add a new column called `height * 100` so specifying the name is important.

6.3.5 arrange

This is one of the nicest R functions in this library, in my (the Professor's) opinion:

1. In base R we say statements like: `dat[order(dat$name), ]` to sort a column, by default it sorts by alphabetical increasing order, though we can also do the reverse. But in `tibble` we use *functions* instead.
   The `tibble` equivalents are: `arrange(dat, name)`, by default it sorts by alphabetical increasing order, though `arrange(dat, desc(name))` does the reverse. You can even decide how to break ties by giving multiple columns: `arrange(dat, gender, height)`.

6.3.6 summarise

Note that the british spelling `summarize` is actually valid as well.

1. In base R we say statements like: `mean(dat$height)`.
   The `tibble` equivalents are: `summarise(dat, mean(height))` or `summarise(dat, avg=mean(height))` for a nicer name.

6.3.7 group_by

`group_by(dat, col)` by itself does not do much, it just adds a line saying it is keeping track of a group. But things get interesting when we combine `group_by` with `summarize`

6.4 Pipe operators

There are 2 pipe operators: $\% > \%$ and $|>$ which means:

$$f(x, y, z) \iff x \mid> f(y, z) \iff x\% > \%f(y, z)$$

This can be used as such:

```r
dat |> filter(gender == "female") |> summarise(avg = mean(height))
```

though often it will be styled as such:

```r
dat |>
    filter(gender == "female")  |>
    summarise(avg = mean(height))
```

7 Wednesday, October 4th

8 Wednesday, October 11th

`runif(n, min, max)` creates random deviates of the uniform distribution

8.1 Programming Structures

In the past, we have seen obj[] and func() but now we will see {} for Programming Structures.

8.2 Compound Expressions

You can internal assignments in compound expressions. If you say

```
A = {
  print("hello")
  b=sqrt(100)
  1:10
}
```

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

Only assigned expressions can be referenced outside (and after) the compound expression.

8.3 If/Else

Unlike some other languages, R always requires braces.

The `else` also needs to be on the same line as the closing brace of the `if` – unless you surround the entire if-else with braces (so that you can treat it as one singular expression).

`else if` is another way to avoid having to have the braces.

Note that we will get to functions 2 weeks from now.

This will error:

```
if (y < 0) {
  print("negative")
}
```

is equiv to

```
if (y < 0) {
  print("negative")
} else {
  NULL
}
```

8.3.1 Vectorized ifelse

The above error can be solved using the vectorized `ifelse(cond, )`

```
ifelse(y < 0, yes="negative", no="non-negative")
```

8.3.2 Testing for missing values

Do NOT do:

```
z = NA
if (z == NA) {
    print("missing")
}
```

as it will error since `z == NA` is just `NA` which is not one of `TRUE` or `FALSE`. Instead, you *should* do:

```
z = NA
if (is.na(z)) {
    print("missing")
}
```

8.4 case_when

Something more practical:
R has a builtin `mtcars` datatset.

If we wanted to assign the mapping:

$$4 - \text{cyl} = \text{"sm"}$$
$$6 - \text{cyl} = \text{"md"}$$
$$8 - \text{cyl} = \text{"lg"}$$

```
dat |>
    mutate(size=case_when(
        cyl == 4 ~ "sm",
        cyl == 6 ~ "md",
        cyl == 8 ~ "lg"
    ))
```

8.5 switch

```
switch(dat$cyl[1],
        4 = "sm",
        6 = "md",
        8 = "lg"
)
```

8.6 Lookup vector trick

```
lookup = c('4' = 'sm', '6' = 'md', '8' = 'lg')
lookup[dat$cyl]
```

9 Wednesday, October 18th

9.1 Control Structures in R: Loops and Iterations

Control structures like loops are used to control the flow of execution in a script. In R, one of the most commonly used loops is the for loop.

9.2 For loops

The for loop allows us to repeat a set of commands for a specific number of times.

```r
for (i in 1:5) {
  print(i)
}
```

This code will print the numbers 1 through 5, once for each iteration of the loop.

9.2.1 Usage with vectors and lists

for loops can also iterate over vectors and lists.

```r
fruits <- c("apple", "banana", "cherry")
for (fruit in fruits) {
  print(fruit)
}
```

Here, the loop iterates over each element of the fruits vector, printing the name of the fruit in each iteration.

9.2.2 Using the seq() function

The seq() function can be useful when you want to iterate over a sequence that doesn't increment by 1.

```r
for (i in seq(1, 10, 2)) {
  print(i)
}
```

This will print out the odd numbers between 1 and 10.

9.3 Control statements within loops

9.3.1 next

The next statement allows you to skip the rest of the current iteration and move on to the next one.

```r
for (i in 1:5) {
  if (i == 3) {
    next
  }
  print(i)
}
```

In this loop, the number 3 will not be printed because when i equals 3, the next statement will skip the rest of the loop body for that iteration.

9.3.2 break

The break statement allows you to exit the loop entirely.

```r
for (i in 1:5) {
  if (i == 4) {
    break
  }
  print(i)
}
```

In this example, the numbers 1, 2, and 3 will be printed. However, when i equals 4, the break statement will be executed, and the loop will terminate, so the numbers 4 and 5 will not be printed.

9.4 Apply Family Functions

The apply family of functions provides a suite of utilities to work on vectors, matrices, data frames, and lists without the need for explicit loops. Each function in this family is specialized to handle a specific data type and returns data in a specific format.

9.4.1 apply()

The apply function is primarily used for matrices or multi-dimensional arrays. It applies a function to the rows or columns of a matrix (or, more generally, to the margins of an array).

- **Syntax:** `apply(X, MARGIN, FUN, ...)`
  - `X`: the array or matrix
  - `MARGIN`: a vector indicating which margins should be "retained". 1 indicates rows, 2 indicates columns.
  - `FUN`: the function to be applied
  - `...`: additional arguments to be passed to the function
- **Example:** To compute the mean of each column in a matrix:

```r
apply(X = dat, MARGIN = 2, FUN = mean)
```

9.5 Anonymous Functions

In R, functions can be created on the fly without naming them—these are called anonymous functions. They are particularly useful when using functions like those in the apply family where you might only need the function temporarily.

9.5.1 Usage in apply()

Anonymous functions can be used directly within the apply function (or any of its relatives). The general structure is:

```r
apply(data, margin, function(variable) expression_using_variable)
```

- **Example:** To compute the range (difference between max and min) of each column in a matrix:

  ```r
  apply(dat, 2, function(x) max(x) - min(x))
  ```

  Here, the anonymous function calculates the difference between the maximum and minimum of x, where x represents each column in the matrix dat.

9.6 Range Computation Using `apply`

To compute the range (difference between maximum and minimum values) for each column of a dataset, we can define a custom function called `RANGE` and then utilize the `apply` function to compute it for each column.

```r
RANGE = function(x, na.rm = FALSE) {
    max(x, na.rm = na.rm) - min(x, na.rm = na.rm)
}
```

```r

apply(dat2, 2, RANGE, na.rm = TRUE)
```

9.7 summarise

The summarise function in the dplyr package of the tidyverse is used to compute summary statistics of a given dataset. It's especially powerful when combined with the group_by function to obtain grouped summaries.

- **Basic Usage:** Summarise allows you to compute summary statistics such as means, medians, or custom aggregations.

```r
    summarise(dat, avg_height = mean(height), total_count = n())
```

- **With Grouping:** Using group_by before summarise lets you calculate summaries for each level of a categorical variable.

```r
dat %>%
    group_by(gender) %>%
    summarise(avg_height = mean(height), total_count = n())
```

- **Multiple Summaries:** You can compute multiple summary statistics within a single call to summarise.

```r
    summarise(dat, avg_height = mean(height), median_height =
     ↪  median(height))
```

- **Using Custom Functions:** You can use custom functions within summarise to calculate desired metrics.

```r
    summarise(dat, range_height = max(height) - min(height))
```

- **NA Handling:** You can manage missing values by using the na.rm argument in your summary functions.

```r
    summarise(dat, avg_height = mean(height, na.rm = TRUE))
```

Note: The summarise function returns a new dataframe where each row represents a summarised value. The original dataframe remains unchanged.

9.8 while loops in R

The 'while' loop in R is used to execute a block of code repeatedly as long as a specified condition is 'TRUE'.

- **Basic Structure:** The structure of a 'while' loop starts with the 'while' keyword, followed by the condition in parentheses, and then the code to be executed within curly braces.

```r
    while (condition) {
        # code to execute
    }
```

- **Example:** A simple counter.

```r
count <- 1
while (count <= 5) {
    print(paste("Count is:", count))
    count <- count + 1
}
```

- **Caution:** Ensure that the loop condition will eventually turn ‘FALSE‘ to avoid infinite loops.
- **Breaking out of a loop:** The ‘break‘ statement can be used to exit a ‘while‘ loop prematurely.

```r
count <- 1
while (TRUE) {
    if (count > 5) {
        break
    }
    print(paste("Count is:", count))
    count <- count + 1
}
```

- **Skipping an iteration:** The ‘next‘ statement allows you to skip the current iteration and proceed to the next one.

```r
count <- 1
while (count <= 5) {
    if (count == 3) {
        count <- count + 1
        next
    }
    print(paste("Count is:", count))
    count <- count + 1
}
```

Note: While ‘while‘ loops offer flexibility, they can be prone to errors like infinite loops. Always ensure your condition will eventually become ‘FALSE‘, or have a mechanism (like ‘break‘) to exit the loop.

10 Wednesday, October 25th

10.1 Variables in R

In R, variables can be broadly classified into global and local variables.

10.1.1 Global and Local Variables

Global variables exist in the global environment and can be accessed from anywhere in the script. On the other hand, local variables are only accessible within their respective functions or scopes.

```r
tortilla = 8
carne = 7
make_taco = function(ingred1, ingred2) {
  ingred1 + ingred2
}
make_taco(tortilla, carne)
```

In the code above, `tortilla` and `carne` are global variables, while `ingred1` and `ingred2` are local variables that act as arguments for the `make_taco()` function.

10.1.2 Lexical Scoping in R

R uses lexical scoping, which refers to determining the value of a variable based on how functions are defined, not when they are invoked. The term "lexical" denotes that scoping rules use a parse-time structure rather than a run-time structure.

Scoping is the process of determining the value of a variable. A variable's scope is the section of the code where it exists and can be accessed.

R's lexical scoping follows four main rules:

- Name masking
- Functions versus variables
- A fresh start
- Dynamic lookup

Local variables cannot be accessed from outside their function. Attempting to do so will result in an error message: "Error in eval(expr, envir, enclos): object '<object name>' not found".

Name Masking Names defined inside a function mask those defined outside. If a name isn't found inside a function, R will look one level up. The same rules apply when a function is nested inside another function.

Functions vs. Variables If a function and a non-function have the same name (but exist in different environments), R can distinguish between them.

Fresh Start Local variables are reset every time a function is called.

Dynamic Lookup Dynamic lookup is R's mechanism for determining values when a function is executed, rather than when it's created.

10.2 Names and Values in R

10.2.1 Understanding Variable Bindings

Consider the following code:

```r
a <- c(1, 2, 3)
```

Here, the variable `a` is bound to the numeric vector $\{1, 2, 3\}$.

```r
typeof(a) # data type
lobstr::obj_size(a) # size
lobstr::obj_addr(a) # memory location
lobstr::ref(a) # tree of references
```

When two variables point to the same data, such as:

```r
a <- c(1, 2, 3)
b <- a
```

The variable `b` does not create a copy of the data but references the same vector as `a`.

10.2.2 Copy on Modify

If the value of one of the objects is modified, R creates a copy of the original data for that object. This is evident when we modify `a` but not `b`.

```r
a[1] <- 0
```

Now, `a` is $\{0, 2, 3\}$ and `b` remains $\{1, 2, 3\}$.

R uses a copy-on-modify (if necessary) strategy. Copying is only done when modifying the object. If modifications don't affect non-local variables, R can modify the object without copying.

10.2.3 Binding Counting in R

R uses a counting heuristic to track the number of names bound to a variable. It classifies bindings into:

- 0 - No bindings.
- 1 - One binding.
- 2+ - Multiple bindings (R doesn't distinguish the exact number beyond two).

When a name is unbound using the `rm()` function, R does not immediately free up the associated memory. Instead, R's garbage collection mechanism manages memory allocation.

10.3 Memory Management in R

R uses a Garbage Collection (GC) mechanism for memory management. When R detects an unbound name, it doesn't immediately free up the memory space. The `rm()` function only removes the binding, but the memory is managed by R's GC.

```r
rm(b)
```

Thus, while the `rm()` function removes the binding between `b` and the numeric vector, it doesn't automatically free the memory space occupied by the numeric vector.

10.4 Advanced R Concepts

10.5 R's Motto: Everything is an Object

In R, everything that exists is considered an object, and all operations are a result of function calls. This includes arguments, values, and even the function and call themselves.

10.5.1 Objects and Functions in R

R has a unique way of treating operations. For example:

- Assignment:

```r
w <- c(2, 4, 6)
# This can be seen as:
"<-"(w, c(2, 4, 6))
w
## [1] 2 4 6
```

- Extraction:

```r
w[2]
# Is equivalent to:
"["(w, 2)
## [1] 4
```

- Replacement:

```r
w[1] <- 0
# Can be represented as:
w = "[<-"(w, 1, 0)
w
## [1] 0 4 6
```

- Arithmetic Operations:

```r
2 + 3
# Is actually a call to:
"+"(2, 3)
## [1] 5
```

11 Wednesday, November 1st

11.1 Environments in R

Environments are key data structures in R that power scoping. They play a significant role in understanding lexical scoping, namespaces, and various evaluation aspects.

11.1.1 Understanding Environments

You can use the `rlang` package to explore and manipulate environments. For instance, the `env()` function from `rlang` can be used to create an environment:

```r
e1 <- env(
  a = c(2, 4, 6),
  b = TRUE,
  c = "hi",
  d = 3.3
)
```

11.1.2 Properties of Environments

Environments share some similarities with named lists, but with distinct features:

- Every name in an environment must be unique.
- The names in an environment are not ordered.
- Every environment has a parent.
- Environments are not duplicated when modified.

To view the content of an environment, rather than just its memory address, you can use the `env_print()` function:

```r
env_print(e1)
## <environment: 0x7fb372f04550>
## parent: <environment: global>
## bindings:
## * a: <dbl>
## * b: <lgl>
## * c: <chr>
## * d: <dbl>
```

This behavior is similar to the distinction between `__str__` and `__repr__` in Python.

11.2 Characteristics of Environments

- Objects within environments are modified in-place, eliminating the need to create copies.
- Every environment has a parent, which is another environment. This parent-child relationship between environments is foundational to R's implementation of lexical scoping: if a name isn't found in an environment, R will search its parent, and so on.
- The only environment without a parent is the empty environment, denoted as `R_EmptyEnv`.
- Every environment's ancestry eventually leads to the empty environment.
- The Global Environment is a special environment. Its ancestors include every attached package.
- When a package is attached using `library()` or `require()`, it becomes a parent of the global environment. The immediate parent of the global environment is the most recently attached package.
- The sequence in which packages are attached is termed the "search path". You can view these environments using `base::search()` or see the environments directly with `rlang::search_envs()`.

11.3 Environments and Function Execution

Every time a function is invoked in R, a new environment, termed the "execution environment," is created to host its execution.

- The parent of the execution environment is the environment of the function, known as the "function environment."
- Execution environments are typically ephemeral. Once the function finishes its execution, its environment is eligible for garbage collection.

11.3.1 Role and Importance of Environments

- During the execution of an expression in R, there's always a local, current, or active environment.
- R spawns a new environment each time a function is executed. Given that everything in R happens as a result of a function call, environments are frequently created and discarded.
- Functions in R often invoke other functions, leading to the creation and disposal of multiple environments in a nested manner.

11.3.2 Function Evaluation in R

Function calls in R are processed in three key steps:

1. The expressions of the arguments in the call (actual arguments) are matched to the formal arguments defined in the function.
2. A fresh environment is created. Within this environment, assignments are made for each formal argument, either containing the actual argument (if provided) or the default value (if the argument is missing and a default exists). The enclosing environment of this newly created environment is the environment where the function resides.

3. The body of the function is evaluated within this new environment. The result of this evaluation is returned as the function's output.

It's crucial to note a significant distinction in R's function evaluation process: the incorporation of the function's environment. This environment, which is where the function was originally defined, influences how names are resolved during execution. Consequently, R searches for names within this function environment.

12 Wednesday, November 8th

Everything that happens in R is an object. This is a foundational principle that influences how R handles computations and data structures. For example, when you perform an operation like `2 + 3`, it is actually interpreted as a function call.

`2 + 3` is the same as `‘+‘(2, 3)`

Tree Perspective of Expressions

Expressions in R can be represented as trees, where nodes represent operations and leaves represent operands or values. This tree structure helps in understanding how R parses and evaluates expressions.

Example 1: Simple Addition

Consider the expression `a = 2 + 3`. Its tree representation is as follows:

```r
# Assigning the sum of 2 and 3 to a
a <- 2 + 3
```

```
    =
   / \
  a   +
     / \
    2   3
```

This tree shows the assignment operation (‘=‘) with ‘a‘ as the left operand and the addition operation (‘+‘) as the right operand. The addition operation itself has ‘2‘ and ‘3‘ as its operands.

Example 2: Nested Addition

In a more complex expression like `1 + 2 + 3`, the tree structure helps us understand the order of operations:

```r
# Adding 1, 2, and 3 sequentially
1 + 2 + 3
```

```
    +
   / \
  +   3
 / \
1   2
```

This tree indicates that R first adds 1 and 2, then adds the result to 3.

Example 3: Parentheses Expressions

Parentheses can alter the default order of operations. For example, in `1 + (2 + 3)`, the addition inside the parentheses is performed first:

```
# Addition with parentheses
1 + (2 + 3)
```

```
    +
   / \
  1   +
     / \
    2   3
```

This tree demonstrates that the expression ‘2 + 3‘ is evaluated first, followed by the addition of 1 to the result.

Understanding the object-oriented nature of R and how expressions are parsed into tree structures can greatly enhance your ability to write and debug R code. This perspective is especially useful for understanding complex expressions and ensuring accurate calculations.

13 Wednesday, November 15 to Wednesday, November 29

13.1 Creating R Packages in RStudio

Initiating the development of a new R package in RStudio is straightforward. Begin by selecting `File` from the top menu, followed by `New Directory` and then choosing **R Package**.

For detailed guidance on package creation, refer to the **Writing R Extensions** manual.

13.2 Essential Components of an R Package

An R package typically consists of the following six key elements:

1. `DESCRIPTION` file: Contains package metadata, serving as its 'business card.'
2. `NAMESPACE` file: Lists the functions exposed to the user.
3. `R/` directory: Stores the R script files containing your package's functions.
4. `man/` directory: Contains Rd files, which are the technical documentation for your package's functions.
5. `[filename].Rproj` file: An RStudio project file, streamlining package use within RStudio.
6. `.Rbuildignore` file: A hidden file specifying what to exclude during package building.

13.3 Rapid Package Development with devtools

Devtools simplifies the initial package development process. Using its 'hello world' template, you can swiftly create a basic package. This template includes a single function, `hello()`, that outputs a greeting.

Within `R/hello.R`, you'll find comments outlining the development process, including key shortcuts:

- Build and Reload Package: Cmd + Shift + B
- Check Package: Cmd + Shift + E
- Test Package: Cmd + Shift + T

Navigate to the Build tab in RStudio to access additional tools like Install and Restart, and Check, along with more build options.

To deploy your package, execute the build and check steps:

1. Build and Reload: Cmd + Shift + B
2. Check: Cmd + Shift + E

Upon successful completion, you will have created a functional 'hello world' R package. Load and test it using:

```
library(hello)
hello()
```

Expected output:

```
## [1] "Hello, world!"
```

13.4 Understanding R Package Structures

R packages can be developed under three main scenarios:

1. A basic, minimal structure.
2. The default structure, typically generated by “devtools”.
3. An advanced, comprehensive structure with additional components.

Regardless of the scenario, every R package must include certain core files: `DESCRIPTION`, `NAMESPACE`, `R/` directory, and `man/` files. Without these, a package is considered invalid. The default structure created using “devtools” adds a `.Rbuildignore` file and a `[filename].Rproj` file. For a more elaborate package, including a `README`, `vignettes`, and `inst/` directories is common practice.

13.5 Crucial Files: DESCRIPTION and NAMESPACE

13.5.1 The DESCRIPTION File

The DESCRIPTION file, crucial in package metadata, adheres to the Debian Control File format. An exemplary structure is as follows:

```
Package: cointoss
Type: Package
Title: Simulates Tossing a Coin
Version: 0.1.0
Author: Gaston Sanchez
Maintainer: Gaston Sanchez <gaston@email.com>
Description: Functions to create a coin object, to toss a coin
multiple times, and to summarize and visualize frequencies
of the tosses.
License: GPL-2
Encoding: UTF-8
LazyData: true
```

This file must contain several mandatory fields, including Package, Title, Description, Author, Maintainer, Version, and License.

13.5.2 The NAMESPACE File

The NAMESPACE file, essential for managing function accessibility, includes namespace directives for imports and exports. Function documentation is typically written using Roxygen comments. For instance:

```r
#' @title Coin toss function
#' @description Simulates tossing a coin a given number of times
#' @param x coin object (a vector)
#' @param times number of tosses
#' @param prob vector of probabilities for each side of the coin
#' @return vector of tosses
toss <- function(x, times = 1, prob = NULL) {
    sample(x, size = times, replace = TRUE, prob = prob)
}
```

13.6 Lifecycle of an R Package

An R package can exist in several states during its development and usage:

1. Source
2. Bundled
3. Binary
4. Installed
5. In-memory

13.6.1 Source Package

This initial state includes all the directories and files in their raw form.

13.6.2 Bundled Package

Next, the source package may be compressed into a `.tar.gz` file, creating a bundled package.

13.6.3 Binary Package

Binary packages are platform-specific compressed files, created from the bundled package.

13.6.4 Installed Package

When a binary package is decompressed and set up in a library, it becomes an installed package.

13.6.5 In-memory Package

Finally, an installed package must be loaded into memory using the `library()` function to be utilized.

13.6.6 Development Strategy

Always initiate package development from the source, transitioning through these stages to the final, installed version.

13.7 The Packaging Workflow in R

Creating a package in R involves several steps, each handling a different aspect of the package's development. You can break down the process into distinct parts, each of which can be addressed separately. A typical packaging workflow includes:

1. Creating documentation
2. Checking documentation
3. Running tests
4. Knitting vignettes
5. Building the package bundle
6. Installing the package
7. Performing a comprehensive check

To facilitate these tasks, the `\devtools"` package offers functions for each action:

```
devtools::document() # Create Documentation
devtools::check_man() # Check Documentation
devtools::test() # Run Tests
devtools::build_vignettes() # Knit Vignettes
devtools::build() # Build Bundle
devtools::install() # Install binary
devtools::check() # Check
```

13.7.1 Detailed Packaging Steps

Create Documentation: Update the manual documentation by regenerating .Rd files in the `man/` directory using `devtools::document()` after modifying roxygen comments in your functions.

Check Documentation: After generating new documentation, it's crucial (and mandatory for CRAN submission) to verify its correctness with `devtools::check_man()`.

Run Tests: For packages with unit-tests in the `tests/` directory, use `devtools::test()` to execute these tests.

Build Vignettes: If your package includes vignettes in the `vignettes/` directory, create them using `devtools::build_vignettes()`. This function knits the `.Rmd` files, producing output in the `inst/` folder.

Build Bundle: Convert your package source into a single `.tar.gz` file using `devtools::build()`. Note that this function doesn't generate or check documentation and doesn't run tests, but it does build vignettes by default.

Install: Install your package (source, bundle, or binary) using `devtools::install()`, making it ready to be loaded with `library()`.

Check: For an extensive review, especially before sharing on CRAN, use `devtools::check()` to perform a comprehensive check of your package.

13.7.2 Utilizing an Optional devtools File

To remember the specific functions for each step, consider including an auxiliary `.R` file named `devtools-flow.R` in your source package, containing the following workflow:

```r
# =====================================================
# Devtools workflow
# =====================================================
devtools::document()         # generate documentation
devtools::check_man()        # check documentation
devtools::test()             # run tests (if any)
devtools::build_vignettes()  # build vignettes (if any)
devtools::build()            # build bundle
devtools::install()          # install package
devtools::check()            # comprehensive check (optional)
```

Exclude this file from the build process by adding its name to the `.Rbuildignore` file, using the caret character (^) as an anchor. Your `.Rbuildignore` file should look something like this:

```
^.*\.Rproj$
^\.Rproj\.user$
^devtools-flow.R
```