R Basics

Fundamental Techniques in Data Science

Kyle M. Lang

Department of Methodology & Statistics
Utrecht University

Outline

Attribution

This course was originally developed by Gerko Vink. You can access the original version of these materials on Dr. Vink's GitHub page: `https://github.com/gerkovink/fundamentals`. The course materials

have been (extensively) modified. Any errors or inaccuracies introduced via these modifications are fully my own responsibility and shall not be taken as representing the views and/or beliefs of Dr. Vink. You can see

Gerko's version of the course on his personal website: `https://www.gerkovink.com/fundamentals`.

Prerequisite Knowledge

After completeing the preperatory exercieses, you should already be familiar with the following ideas.

- What is R?
- What is RStudio?
- Basic R data objects
  - Atomic Vectors
  - Matrices
  - Lists
  - Data Frames
  - Factors
- Visualization with Base R graphics

As part of this week's lab exercises, you will complete detailed tutorials covering:

- How to create reproducible reports with Quarto
- How to load data from external files

We will not cover these topics in the lectures.

FUNCTIONS

R Functions

Functions are the foundation of R programming.

- Other than data objects, almost everything else that you interact with when using R is a function.
- Any R command written as a word followed by parentheses, `()`, is a function.
    - `mean()`
    - `library()`
    - `mutate()`
- Infix operators are aliased functions.
    - `<-`
    - `+`, `-`, `*`
    - `>`, `<`, `==`

User-Defined Functions

We can define our own functions using the `function()` function.

```
square <- function(x) {
    out <- x^2
    out
}
```

After defining a function, we call it in the usual way.

```
square(5)

[1] 25
```

One-line functions don't need braces.

```
square <- function(x) x^2
square(5)

[1] 25
```

User-Defined Functions

Function arguments are not strictly typed.

```
square(1:5)

[1]  1  4  9 16 25

square(pi)

[1] 9.869604

square(TRUE)

[1] 1
```

But there are limits.

```
square("bob") # But one can only try so hard

Error in x^2:  non-numeric argument to binary operator
```

User-Defined Functions

Functions can take multiple arguments.

```
mod <- function(x, y) x %% y
mod(10, 3)

[1] 1
```

Sometimes it's useful to specify a list of arguments.

```
getLsBeta <- function(datList) {
    X <- datList$X
    y <- datList$y

    solve(crossprod(X)) %*% t(X) %*% y
}
```

User-Defined Functions

```
X       <- matrix(runif(500), ncol = 5)
datList <- list(y = X %*% rep(0.5, 5), X = X)

getLsBeta(datList = datList)

     [,1]
[1,]  0.5
[2,]  0.5
[3,]  0.5
[4,]  0.5
[5,]  0.5
```

User-Defined Functions

Functions are first-class objects in R.

- We can treat functions like any other R object.

R views an unevaluated function as an object with type ”closure”.

```
class(getLsBeta)

[1] "function"

typeof(getLsBeta)

[1] "closure"
```

An evaluated functions is equivalent to the objects it returns.

```
class(getLsBeta(datList))

[1] "matrix" "array"

typeof(getLsBeta(datList))

[1] "double"
```

Nested Functions

We can use functions as arguments to other operations and functions.

```r
fun1 <- function(x, y) x + y

## What will this command return?
fun1(1, fun1(1, 1))
```

```
[1] 3
```

Why would we care?

```r
s2 <- var(runif(100))
x  <- rnorm(100, 0, sqrt(s2))
```

Nested Functions

```
X[1:8, ]
```

```
          [,1]       [,2]       [,3]      [,4]       [,5]
[1,] 0.52431382 0.67136447 0.28228726 0.7148383 0.54204681
[2,] 0.01926742 0.11693762 0.09148502 0.6929171 0.88371944
[3,] 0.05100735 0.18432074 0.43547799 0.6097462 0.09026598
[4,] 0.60566972 0.12944127 0.21000143 0.2441917 0.68141473
[5,] 0.48737303 0.94030405 0.23988619 0.4915910 0.36353771
[6,] 0.19941958 0.96670678 0.11455820 0.1243947 0.24253273
[7,] 0.95507804 0.38705829 0.49733535 0.2968470 0.81001800
[8,] 0.11093197 0.07731757 0.84923006 0.8653987 0.61914193
```

```
c(1, 3, 6:9, 12)
```

```
[1]  1  3  6  7  8  9 12
```

ITERATION

Loops

There are three types of loops in R: *for*, *while*, and *until*.

- You'll rarely use anything but the for loop.
- So, we won't discuss while or until loops.

A *for loop* is defined as follows.

```
for(INDEX in RANGE) { Stuff To Do with the Current INDEX Value }
```

Loops

For example, the following loop will sum the numbers from 1 to 100.

```
val <- 0
for(i in 1:100) {
    val <- val + i
}

val

[1] 5050
```

Loops

This loop will compute the mean of every column in the `mtcars` data.

```
means <- rep(0, ncol(mtcars))
for(j in 1:ncol(mtcars)) {
    means[j] <- mean(mtcars[ , j])
}

means

 [1]  20.090625   6.187500 230.721875 146.687500   3.596563
 [6]   3.217250  17.848750   0.437500   0.406250   3.687500
[11]   2.812500
```

Loops

Loops are often one of the least efficient solutions in R.

```
n <- 1e8

t0 <- system.time({
    val0 <- 0
    for(i in 1:n) val0 <- val0 + i
})

t1 <- system.time(
    val1 <- sum(1:n)
)
```

Loops

Both approaches produce the same answer.

```
val0 - val1

[1] 0
```

But the loop is many times slower.

```
t0

   user  system elapsed 
  1.352   0.000   1.352 

t1

   user  system elapsed 
      0       0       0 
```

Loops

There is often a built in routine for what you are trying to accomplish with the loop.

```
## The appropriate way to get variable means:
colMeans(mtcars)

       mpg        cyl       disp         hp       drat
 20.090625   6.187500 230.721875 146.687500   3.596563
        wt       qsec         vs         am       gear
  3.217250  17.848750   0.437500   0.406250   3.687500
      carb
  2.812500
```

Apply Statements

In R, some flavor of *apply statement* is often preferred to a loop.

- Apply statements broadcast some operation across the elements of a data object.
- Apply statements can take advantage of internal optimizations that loops can't use.

There are many flavors of apply statement in R, but the three most common are:

- `apply()`
- `lapply()`
- `sapply()`

Apply Statements

Apply statements generally take one of two forms:

```
apply(DATA, MARGIN, FUNCTION, ...)

apply(DATA, FUNCTION, ...)
```

Apply Examples

```
## Load some example data:
data(mtcars)

## Subset the data:
dat1 <- mtcars[1:5, 1:3]

## Find the range of each row:
apply(dat1, 1, range)

     Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive
[1,]         6             6          4              6
[2,]       160           160        108            258
     Hornet Sportabout
[1,]                 8
[2,]               360
```

Apply Examples

```
## Find the maximum value in each column:
apply(dat1, 2, max)

  mpg   cyl  disp
 22.8   8.0 360.0

## Subtract 1 from every cell:
apply(dat1, 1:2, function(x) x - 1)

                   mpg cyl disp
Mazda RX4         20.0   5  159
Mazda RX4 Wag     20.0   5  159
Datsun 710        21.8   3  107
Hornet 4 Drive    20.4   5  257
Hornet Sportabout 17.7   7  359
```

Apply Examples

```r
## Create a toy list:
l1 <- list()
for(i in 1:3) l1[[i]] <- runif(10)

## Find the mean of each list entry:
lapply(l1, mean)

[[1]]
[1] 0.526697

[[2]]
[1] 0.4020885

[[3]]
[1] 0.607818

## Same as above, but return the result as a vector:
sapply(l1, mean)

[1] 0.5266970 0.4020885 0.6078180
```

Apply Examples

```
## Find the range of each list entry:
lapply(l1, range)

[[1]]
[1] 0.04395916 0.99350611

[[2]]
[1] 0.002797563 0.821082495

[[3]]
[1] 0.09926892 0.90430843

sapply(l1, range)

           [,1]        [,2]       [,3]
[1,] 0.04395916 0.002797563 0.09926892
[2,] 0.99350611 0.821082495 0.90430843
```

Apply Examples

We can add additional arguments needed by the function.

- These arguments must be named.

```
apply(dat1, 2, mean, trim = 0.1)

   mpg    cyl   disp
 20.98   6.00 209.20

sapply(dat1, mean, trim = 0.1)

   mpg    cyl   disp
 20.98   6.00 209.20
```

DATA MANIPULATION

Base R Subsetting

In Base R, we typically use three operators to subset objects:

- `[]`
- `[[]]`
- `$`

Which of these operators we choose to use (and how we implement the chosen operator) will depend on two criteria:

- What type of object are we trying to subset?
- How much of the original typing do we want to keep in the subset?

Tidyverse Subsetting

The **dplyr** package provides many ways to subset data, but two functions are most frequently useful.

- `select()` : subset columns
- `filter()` : subset rows

```
library(dplyr)
```

Subsetting Columns: `select()`

The `dplyr::select()` function provides a very intuitive syntax for variable selection and column-wise subsetting.

```
select(d3, a, b)

Error:  object 'd3' not found
```

```
select(d3, -a)

Error:  object 'd3' not found
```

Subsetting Rows

The `dplyr::filter()` function provides easy row subsetting:

```
filter(d3, c > 0.5)

Error:  object 'd3' not found
```

```
filter(d3, c > 0.15, b == "foo")

Error:  object 'd3' not found
```

We can achieve the same effect via logical indexing in Base R:

```
d3[d3$c > 0.5, ]

Error:  object 'd3' not found
```

```
d3[d3$c > 0.15 & d3$b == "foo", ]

Error:  object 'd3' not found
```

Base R Variable Transformations

There is nothing very special about the process of transforming variables in Base R.

```
d4   <- d3
Error:  object 'd3' not found
d4$d <- scale(d4$c)
Error:  object 'd4' not found
d4$e <- !d4$a
Error:  object 'd4' not found
d4
Error:  object 'd4' not found
```

```
d4   <- d3
Error:  object 'd3' not found
d4$c <- scale(d4$c, scale = FALSE)
Error:  object 'd4' not found
d4$a <- as.numeric(d4$a)
Error:  object 'd4' not found
d4
Error:  object 'd4' not found
```

Tidyverse Variable Transformations

The `mutate()` function from **dplyr** is the workhorse of Tidyverse transformation functions.

```
mutate(d3, d = rbinom(nrow(d3), 1, c))

Error:  object 'd3' not found
```

```
mutate(d3,
       d = rbinom(nrow(d3), 1, c),
       e = d * c
       )

Error:  object 'd3' not found
```

Sorting & Ordering

To sort a single vector, the best option is the Base R `sort()` function.

```
sort(d3$c)

Error:  object 'd3' not found

sort(d3$c, decreasing = TRUE)

Error:  object 'd3' not found
```

To sort the rows of a data frame according to the order of one of its columns, the `dplyr::arrange()` works best.

- You can use the Base R `order()` function to achieve similar results.
- The behavior of `order()` is (extremely) unintuitive.

Tidyverse Ordering

Using `dplyr::arrange()` could not be simpler.

```
arrange(d3, a)

Error:  object 'd3'
not found
```

```
arrange(d3, -c)

Error:  object 'd3'
not found
```

```
arrange(d3, -a, c)

Error:  object 'd3'
not found
```

PIPES

What are pipes?

The %>% symbol represents the *pipe* operator.

- We use the pipe operator to compose functions into a *pipeline*.

The following code represents a pipeline.

```
firstBoys <-
    readRDS("boys.rds") %>%
    head()
```

This pipeline replaces the following code.

```
firstBoys <- head(readRDS("boys.rds"))
```

Why are pipes useful?

Let's assume that we want to:

1. Load data
2. Transform a variable
3. Filter cases
4. Select columns

Without a pipe, we may do something like this:

```
library(dplyr)

boys <- readRDS("../../../data/boys.rds")

Error in gzfile(file, "rb"):  cannot open the connection

boys <- transform(boys, hgt = hgt / 100)

Error:  object 'boys' not found

boys <- filter(boys, age > 15)

Error:  object 'boys' not found
```

Why are pipes useful?

With the pipe, we could do something like this:

```
boys <-
    readRDS("../../../data/boys.rds") %>%
    transform(hgt = hgt / 100) %>%
    filter(age > 15) %>%
    subset(select = c(hgt, wgt, bmi))

Error in gzfile(file, "rb"):  cannot open the connection
```

With a pipeline, our code more clearly represents the sequence of steps in our analysis.

Benefits of Pipes

When you use pipes, your code becomes more readable.

- Operations are structured from left to right instead of in to out.
- You can avoid many nested function calls.
- You don't have to keep track of intermediate objects.
- It's easy to add steps to the sequence.

In RStudio, you can use a keyboard shortcut to insert the %>% symbol.

- Windows/Linux: *ctrl* + *shift* + *m*
- Mac: *cmd* + *shift* + *m*

What do pipes do?

Pipes compose R functions without nesting.

- `f(x)` becomes `x` `%>%` `f()`

```
mean(rnorm(10))
```

```
[1] 0.09157446
```

```
rnorm(10) %>% mean()
```

```
[1] -0.4914519
```

What do pipes do?

Multiple function arguments are fine.

- `f(x, y)` becomes `x` `%>%` `f(y)`

```
cor(boys, use = "pairwise.complete.obs")

Error:  object 'boys' not found

boys %>% cor(use = "pairwise.complete.obs")

Error:  object 'boys' not found
```

What do pipes do?

Composing more than two functions is easy, too.

- `h(g(f(x)))` becomes `x` `%>%` `f` `%>%` `g` `%>%` `h`

```
max(na.omit(subset(boys, select = wgt)))

Error:  object 'boys' not found

boys %>%
    subset(select = wgt) %>%
    na.omit() %>%
    max()

Error:  object 'boys' not found
```

The Role of `.` in a Pipeline

In the expression `a %>% f(arg1, arg2, arg3)`, `a` will be "piped into" `f()` as `arg1`.

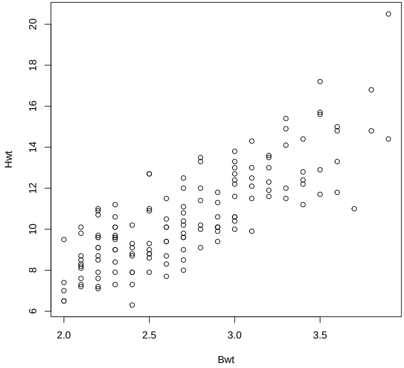
```
data(cats, package = "MASS")
cats %>% plot(Hwt ~ Bwt)

Error in text.default(x, y, txt, cex = cex, font = font):  invalid
mathematical annotation
```

Clearly, we have a problem if we pipe our data into the wrong argument.

- We can change this behavior with the `.` symbol.
- The `.` symbol acts as a placeholder for the data in a pipeline.

The Role of `.` in a Pipeline

```
cats %>% plot(Hwt ~ Bwt, data = .)
```



Exposition Pipe: %$%

There are several different flavors of pipe. The *exposition pipe*, %$%, is a particularly useful variant.

- The exposition pipe *exposes* the contents of an object to the next function in the pipeline.

```
cats %$% plot(Hwt ~ Bwt)

Error in cats %$% plot(Hwt ~ Bwt):  could not find function "%$%"
```

Performing a T-Test in a Pipeline

```
cats %$% t.test(Hwt ~ Sex)
```

```
Error in wrap(., w = 80):  could not find function "wrap"
```

The above is equivalent to either of the following.

```
cats %>% t.test(Hwt ~ Sex, data = .)
t.test(Hwt ~ Sex, data = cats)
```

Workflow & Project Management

Some Programming Tips

You can save yourself a great deal of heartache by following a few simple guidelines.

- Keep your code tidy.
- Use comments to clarify what you are doing.
- When working with functions in RStudio, use the TAB key to quickly access the documentation of the function's arguments.
- Give your R scripts and objects meaningful names.
- Use a consistent directory structure and RStudio projects.

General Style Advice

Use common sense and BE CONSISTENT.

- Browse the tidyverse style guide.
  - The point of style guidelines is to enforce a common vocabulary.
  - You want people to concentrate on *what* you're saying, not *how* you're saying it.
- If the code you add to a project/codebase looks drastically different from the extant code, the incongruity will confuse readers and collaborators.

Spacing and whitespace are your friends.

- `a<-c(1,2,3,4,5)`
- `a <- c(1, 2, 3, 4, 5)`
- At least put spaces around assignment operators and after every comma!

References