

Getting to know R

EC 425/525, Lab 1

Edward Rubin

05 April 2019

Prologue

Schedule

Today

Get to know R

1. Basic features of R
2. Fun with functions
3. OLS (canned and custom)
4. Simulations

Object types/classes

As we discussed in class, R revolves around objects, e.g., `test <- 123.`

Object types/classes

As we discussed in class, R revolves around objects, e.g., `test <- 123`.

Note You can also assign values to objects via `=`, e.g., `test = 123`.

Object types/classes

As we discussed in class, R revolves around objects, e.g., `test <- 123`.

Note You can also assign values to objects via `=`, e.g., `test = 123`.

Objects have types/classes.

Object types/classes

As we discussed in class, R revolves around objects, e.g., `test <- 123`.

Note You can also assign values to objects via `=`, e.g., `test = 123`.

Objects have types/classes.

- `1`, `2/3`, and `are` are `numeric`.

Object types/classes

As we discussed in class, R revolves around objects, e.g., `test <- 123`.

Note You can also assign values to objects via `=`, e.g., `test = 123`.

Objects have types/classes.

- `1`, `2/3`, and `are` are `numeric`.
- `"Hello"` and `'cruel world'` are both `character`.

R intro

Object types/classes

As we discussed in class, R revolves around objects, e.g., `test <- 123`.

Note You can also assign values to objects via `=`, e.g., `test = 123`.

Objects have types/classes.

- `1`, `2/3`, and `are` are `numeric`.
- `"Hello"` and `'cruel world'` are both `character`.
- `TRUE`, `T`, `FALSE`, and `F` are `logical` (as is the result of `3 > 2`).

R intro

Object types/classes

As we discussed in class, R revolves around objects, e.g., `test <- 123`.

Note You can also assign values to objects via `=`, e.g., `test = 123`.

Objects have types/classes.

- `1`, `2/3`, and `are` are `numeric`.
- `"Hello"` and `'cruel world'` are both `character`.
- `TRUE`, `T`, `FALSE`, and `F` are `logical` (as is the result of `3 > 2`).

The `class(x)` function tells you the class of object `x`.

Object types/classes

```
1
```

```
#> [1] 1
```

```
"Clever/funny example words?"
```

```
#> [1] "Clever/funny example words?"
```

```
3 < 2
```

```
#> [1] FALSE
```

```
"Warriors" > "Bucks"
```

```
#> [1] TRUE
```

Object types/classes

```
1
```

```
#> [1] 1
```

```
"Clever/funny example words?"
```

```
#> [1] "Clever/funny example words?"
```

```
3 < 2
```

```
#> [1] FALSE
```

```
"Warriors" > "Bucks"
```

```
#> [1] TRUE
```

```
class(1)
```

```
#> [1] "numeric"
```

```
class("Clever/funny example words?")
```

```
#> [1] "character"
```

```
class(3 < 2)
```

```
#> [1] "logical"
```

```
class("Warriors" > "Bucks")
```

```
#> [1] "logical"
```

Structure

In addition to having types/classes, objects have some type of structure.

- `1:3`, `c(1, 2)`, and `seq(2, 8, 2)` each produce a `numeric`-class `vector`.

Structure

In addition to having types/classes, objects have some type of structure.

- `1:3`, `c(1, 2)`, and `seq(2, 8, 2)` each produce a `numeric`-class `vector`.
- `c("Alright", "already")` produces a `vector` of `character` class.

Structure

In addition to having types/classes, objects have some type of structure.

- `1:3`, `c(1, 2)`, and `seq(2, 8, 2)` each produce a `numeric`-class `vector`.
- `c("Alright", "already")` produces a `vector` of `character` class.
- `c(1, 3, T, "Hello")` produces a `vector` of `character` class.

Structure

In addition to having types/classes, objects have some type of structure.

- `1:3`, `c(1, 2)`, and `seq(2, 8, 2)` each produce a `numeric`-class `vector`.
- `c("Alright", "already")` produces a `vector` of `character` class.
- `c(1, 3, T, "Hello")` produces a `vector` of `character` class.
- `matrix(data = 1:15, ncol = 5)` creates a `matrix` with class from `data`.

Structure

In addition to having types/classes, objects have some type of structure.

- `1:3`, `c(1, 2)`, and `seq(2, 8, 2)` each produce a `numeric`-class `vector`.
- `c("Alright", "already")` produces a `vector` of `character` class.
- `c(1, 3, T, "Hello")` produces a `vector` of `character` class.
- `matrix(data = 1:15, ncol = 5)` creates a `matrix` with class from `data`.
- `data.frame(x = 1:2, y = c("a", "b"), z = T)` produces a `data.frame` with three columns and two rows. The first column (`x`) is `numeric`; the second column (`y`) is `character`, and the third column (`z`) is `logical`.

Object types

Our `matrix`

```
matrix(data = 1:15, ncol = 5)
```

```
#>      [,1] [,2] [,3] [,4] [,5]  
#> [1,]    1    4    7   10   13  
#> [2,]    2    5    8   11   14  
#> [3,]    3    6    9   12   15
```

R intro

Object types

Our `matrix`

```
matrix(data = 1:15, ncol = 5)
```

```
#>      [,1] [,2] [,3] [,4] [,5]  
#> [1,]    1    4    7   10   13  
#> [2,]    2    5    8   11   14  
#> [3,]    3    6    9   12   15
```

Our first `data.frame`!

```
data.frame(x = 1:3, y = T)
```

```
#>   x     y  
#> 1 1 TRUE  
#> 2 2 TRUE  
#> 3 3 TRUE
```

R intro

Object types

Our `matrix`

```
matrix(data = 1:15, ncol = 5)
```

```
#>      [,1] [,2] [,3] [,4] [,5]  
#> [1,]    1    4    7   10   13  
#> [2,]    2    5    8   11   14  
#> [3,]    3    6    9   12   15
```

Our first `data.frame`!

```
data.frame(x = 1:3, y = T)
```

```
#>   x     y  
#> 1 1 TRUE  
#> 2 2 TRUE  
#> 3 3 TRUE
```

Notice how R helps 'fill' out the columns when lengths don't match.

R intro

Object types

R can help you check object's type.

```
class(matrix(1:9, ncol = 3))
```

```
#> [1] "matrix"
```

```
is.matrix(matrix(1:9, ncol = 3))
```

```
#> [1] TRUE
```

```
is.data.frame(matrix(1:9, ncol = 3))
```

```
#> [1] FALSE
```

Object types

R can help you check object's type.

```
class(matrix(1:9, ncol = 3))
```

```
#> [1] "matrix"
```

```
is.matrix(matrix(1:9, ncol = 3))
```

```
#> [1] TRUE
```

```
is.data.frame(matrix(1:9, ncol = 3))
```

```
#> [1] FALSE
```

```
class(data.frame(x = 1:3))
```

```
#> [1] "data.frame"
```

```
is.matrix(data.frame(x = 1:3))
```

```
#> [1] FALSE
```

```
is.data.frame(data.frame(x = 1:3))
```

```
#> [1] TRUE
```

Object types/classes

Q What happens when we mix classes, e.g., `c(12, "B", F)`?

R intro

Object types/classes

Q What happens when we mix classes, e.g., `c(12, "B", F)`?

A R applies the class that can apply to all objects.

```
c(12, "B")
```

```
#> [1] "12" "B"
```

```
c("B", F)
```

```
#> [1] "B"      "FALSE"
```

```
c(12, F)
```

```
#> [1] 12 0
```

```
c(12, "B", F)
```

```
#> [1] "12"      "B"      "FALSE"
```


Changing types and classes

Change numbers to characters.

```
as.character(1:3)
```

```
#> [1] "1" "2" "3"
```

Change logical to numeric.

```
as.numeric(c(T, F))
```

```
#> [1] 1 0
```

Change vector to matrix.

```
as.matrix(1:3)
```

```
#>      [,1]  
#> [1,]  1  
#> [2,]  2  
#> [3,]  3
```

Packages

Straight out of the box, R has a ton of useful features, but it really gets its power from the additional packages (libraries) that users create.

- **Open-source greatness** Users find needs and create amazing solutions.
- ***Caveat utilitor*** There are a lot of packages, each with a lot of functions. Mistakes can happen.
- **Open-source greatness₂** Again, R is open source: Check the code!

Packages

Straight out of the box, R has a ton of useful features, but it really gets its power from the additional packages (libraries) that users create.

- **Open-source greatness** Users find needs and create amazing solutions.
- ***Caveat utilitor*** There are a lot of packages, each with a lot of functions. Mistakes can happen.
- **Open-source greatness₂** Again, R is open source: Check the code! (Maybe. Sometimes it's very hard.)

Packages

Straight out of the box, R has a ton of useful features, but it really gets its power from the additional packages (libraries) that users create.

- **Open-source greatness** Users find needs and create amazing solutions.
- ***Caveat utilitor*** There are a lot of packages, each with a lot of functions. Mistakes can happen.
- **Open-source greatness₂** Again, R is open source: Check the code! (Maybe. Sometimes it's very hard.)

Examples `ggplot2` (plotting), `dplyr` (data work that can link with SQL), `sf` and `raster` (geospatial work), `lfe` (high-dimensional fixed-effect regression), `data.table` (fast and efficient data work)

Installing packages

Once you find a function/package that you need to install,[†] you'll typically install it via `install.packages("newAmazingPackage")`.^{††}

We'll use the package `dplyr` throughout the course. Let's install it.

```
# Install 'dplyr' package  
install.packages("dplyr")
```

[†] Tool #1: Google. ^{††} The quotation marks are important.

Installing packages

Once you find a function/package that you need to install,[†] you'll typically install it via `install.packages("newAmazingPackage")`.^{††}

We'll use the package `dplyr` throughout the course. Let's install it.

```
# Install 'dplyr' package  
install.packages("dplyr")
```

Aside Notice the comment above the actual code (R uses `#` for comments).

[†] Tool #1: Google. ^{††} The quotation marks are important.

Installing packages

Once you find a function/package that you need to install,[†] you'll typically install it via `install.packages("newAmazingPackage")`.^{††}

We'll use the package `dplyr` throughout the course. Let's install it.

```
# Install 'dplyr' package  
install.packages("dplyr")
```

Aside Notice the comment above the actual code (R uses `#` for comments). While not necessary for R to work, comments are necessary for research.

[†] Tool #1: Google. ^{††} The quotation marks are important.

Using packages

Once you install a package, it is on your machine.

You don't need to install it again—though you probably should update them from time to time.

R intro

Using packages

Once you install a package, it is on your machine.

You don't need to install it again—though you probably should update them from time to time.

To **load a package**, use the `library(package)` function[†], e.g., to load `dplyr`

```
# Load 'dplyr'  
library(dplyr)
```

[†] Notice `library()` doesn't *need* quotation marks. I know...

R intro

Using packages

Once you install a package, it is on your machine.

You don't need to install it again—though you probably should update them from time to time.

To **load a package**, use the `library(package)` function[†], e.g., to load `dplyr`

```
# Load 'dplyr'  
library(dplyr)
```

Now all functions contained in `dplyr` are available (until you close R).

[†] Notice `library()` doesn't *need* quotation marks. I know...

Package management

All of this installing, loading, updating, checking-for-existence-and-then-loading can get old.

As can typing `library(pacakge1)`, `library(package2)`, ...

Package management

All of this installing, loading, updating, checking-for-existence-and-then-loading can get old.

As can typing `library(pacakge1)`, `library(package2)`, ...

[Enter] The `pacman` package... for package management, of course.

Package management

All of this installing, loading, updating, checking-for-existence-and-then-loading can get old.

As can typing `library(package1)`, `library(package2)`, ...

[Enter] The `pacman` package... for package management, of course.

After installing (`install.packages("pacman")`), you can

- Install and load packages via `p_load(package1, ..., packageN)`
- Update packages via `p_update()`

The `p_load` paradigm is especially helpful for collaborations or projects across multiple machines.

R intro

Math in R

Basic algebra: scalars `a` and `b`

```
# Addition  
a + b  
# Subtraction  
a - b  
# Multiplication  
a * b  
# Division  
a / b  
# Mod  
a %% b  
# Integer division  
a %/% b  
# Exponents  
a^b
```

R intro

Math in R

Basic algebra: scalars `a` and `b`

```
# Addition  
a + b  
# Subtraction  
a - b  
# Multiplication  
a * b  
# Division  
a / b  
# Mod  
a %% b  
# Integer division  
a %/% b  
# Exponents  
a^b
```

Matrix algebra: matrices `A` and `B`

```
# Addition  
A + B  
# Subtraction  
A - B  
# Multiplication  
A %*% B  
# Inverse  
solve(A)  
# Transpose  
t(A)  
# Diagonal  
diag(A)  
# Dimensions  
dim(A); nrow(A); ncol(A)
```

Vectorization

One **great** feature in R: vectorization.

With vectorization, R automatically applies functions to each element of a vector—no iteration required.

Vectorization

```
# Multiply a scalar by a scalar  
3 * 4
```

```
#> [1] 12
```

```
# Multiply a scalar by a vector  
3 * c(4, 5, 6)
```

```
#> [1] 12 15 18
```

```
# Multiply a vector by a vector  
1:3 * c(4, 5, 6)
```

```
#> [1] 4 10 18
```

Vectorization can be confusing.

```
c(0.5, 0.9) + c(1, 2, 3)
```

```
#> [1] 1.5 2.9 3.5
```

R will send you a warning, but it won't stop you.

R intro

Statistics in R

Summaries for samples `x` and `y`

```
# Mean  
mean(x)  
# Median  
median(x)  
# Std. dev. and variance  
sd(x)  
var(x)  
# Min. and max.  
min(x)  
max(x)  
# Correlation/covariance  
cor(x, y)  
cov(x, y)  
# Quartiles and mean  
summary(x)
```

R intro

Statistics in R

Summaries for samples `x` and `y`

```
# Mean
mean(x)
# Median
median(x)
# Std. dev. and variance
sd(x)
var(x)
# Min. and max.
min(x)
max(x)
# Correlation/covariance
cor(x, y)
cov(x, y)
# Quartiles and mean
summary(x)
```

Sampling

```
# Set the seed
set.seed(246)
# 4 random draws from N(3,5)
rnorm(n = 4, mean = 3, sd = sqrt(5))
# CDF for N(0,1) at z=1.96
pnorm(q = 1.96, mean = 0, sd = 1)
# Sample 5 draws from x w/ repl.
sample(
  x = x,
  size = 5,
  replace = T
)
# First and last 3
head(x, 3)
tail(x, 3)
```

Indexing vectors

Because vectors are so central to R, being able to index your vectors is important. *Note:* Vectors have one dimension.

Take the vector `x` (e.g., `x <- c(2, 4, 6, 9)`).

- `x[3]` will give us the third element of the vector—i.e., `6`.
- `x[2:3]` will give us the second *and* third elements—i.e., `c(4, 6)`.
- `x[-1]` returns all elements *except the first*—i.e., `c(4, 6, 9)`.
- `x[2] <- 0` replaces the second element with `0`—i.e., `c(2, 0, 6, 9)`.

Indexing vectors

Because vectors are so central to R, being able to index your vectors is important. *Note:* Vectors have one dimension.

Take the vector `x` (e.g., `x <- c(2, 4, 6, 9)`).

- `x[3]` will give us the third element of the vector—i.e., `6`.
- `x[2:3]` will give us the second *and* third elements—i.e., `c(4, 6)`.
- `x[-1]` returns all elements *except the first*—i.e., `c(4, 6, 9)`.
- `x[2] <- 0` replaces the second element with `0`—i.e., `c(2, 0, 6, 9)`.

Lists, e.g., `list(1, 2, 3)`, are similar but use double brackets, e.g., `y[[3]]`.

Indexing matrices

Because matrices (and data frames) have two dimensions, we need to index both dimensions.

For matrix `A` (e.g., `A <- matrix(1:9, ncol = 3)`)

- `A[3,1]` references the element in the 3rd row and 1st column.
- `A[3,]` references all elements in the 3rd row (across all columns).
- `A[,1]` references all elements in the 1st column (across all rows).
- `A[-2,]` returns all elements in `A` except for the 2nd row.
- `A[2,3] <- 0` replaces the element `A[2,3]` with zero.

Indexing matrices

Because matrices (and data frames) have two dimensions, we need to index both dimensions.

For matrix `A` (e.g., `A <- matrix(1:9, ncol = 3)`)

- `A[3,1]` references the element in the 3rd row and 1st column.
- `A[3,]` references all elements in the 3rd row (across all columns).
- `A[,1]` references all elements in the 1st column (across all rows).
- `A[-2,]` returns all elements in `A` except for the 2nd row.
- `A[2,3] <- 0` replaces the element `A[2,3]` with zero.

You can also name rows/columns in matrices—and can use these names for referencing.

Other

"Special" values

- `Inf` is ∞ , *i.e.*, $1/0$. `-Inf` is $-\infty$.
- `NA` is missing.
- `NaN` is *not a number*.
- `NULL` is null.

Other

"Special" values

- `Inf` is ∞ , i.e., $1/0$. `-Inf` is $-\infty$.
- `NA` is missing.
- `NaN` is *not a number*.
- `NULL` is null.

Standard logical operators

- `==` for equality
- `!=` is not equal.
- `>`, `>=`, `<`, `<=`
- `&` is *and*; `|` is *or*.

Other

"Special" values

- `Inf` is ∞ , i.e., $1/0$. `-Inf` is $-\infty$.
- `NA` is missing.
- `NaN` is *not a number*.
- `NULL` is null.

Standard logical operators

- `==` for equality
- `!=` is not equal.
- `>`, `>=`, `<`, `<=`
- `&` is *and*; `|` is *or*.

R orders by number, lowercase, then uppercase.

```
# Ordering  
1 < "a"
```

```
#> [1] TRUE
```

R intro

NA

Finally, NA contains no information in R

```
NA = NA
```

```
#> [1] NA
```

```
NA ≠ NA
```

```
#> [1] NA
```

```
NA > 0
```

```
#> [1] NA
```

```
NA + 0
```

```
#> [1] NA
```

```
is.vector(NA)
```

```
#> [1] TRUE
```

Exercises

1. Using the tools we've covered, generate a dataset ($n = 50$) such that

$$y_i = 12 + 1.5x_i + \varepsilon_i$$

where $x_i \sim N(3, 7)$ and $\varepsilon_i \sim N(0, 1)$.

2. Estimate the relationship via OLS using only matrix algebra. Recall

$$\hat{\beta}_{\text{OLS}} = (X'X)^{-1}X'y$$

3. **Harder** Write a function that estimates OLS coefficients using matrix algebra. Compare your results with the canned function from R (`lm`).
4. **Hardest** Bring it all together: Use your DGP (1) and function (3) to run a simulation that illustrates the unbiasedness of OLS.

Table of contents

Introduction to R

1. Schedule
2. Object types and classes
 - Data structures
 - Mixing types/classes
 - Changing
3. Packages
4. Math in R
5. Vectorization
6. Statistics and simulation
7. Indexing
8. NA and logical operators
9. Exercise