R preliminaries

Max Turgeon

STAT 3150–Statistical Computing

Quick Introduction

Why use R?

- Modern statistics relies heavily on statistical computing.
  - Simulation studies
  - Data analysis
- R is a programming language that can be used on most platforms (Mac, Windows, Linux, Solaris, etc.)
- R is very flexible.
  - It can be extended via R packages.
- R offers a powerful interface for analyzing data and producing high-quality plots.
  - Extensive ecosystem of packages (unlike Julia)

Interacting with R

- You can interact with R in many ways:
  - Through the command line
  - In batch mode (i.e. running a script)
  - Through an Integrated Development Interface (IDE)
- I strongly recommend using RStudio, which is the most powerful IDE for R.

R is a functional programming language

- In R, functions are *first-class citizens*:
  - They can be assigned to variables
  - They can be passed as function arguments
  - They can be returned by other functions
- Everything that happens in R is a function call.
  - E.g. Control structures are implemented as functions too!
- Therefore, to become effective in R, learn to write functions

Every R object is a vector

- Even scalars are vectors of length 1
- There are two main types of vectors:
    - **Atomic vectors**: each element is of the same primitive type (e.g. numeric, boolean, character)
    - **Lists**: elements can be of any type, even lists!
- Matrices and arrays are also vectors, but with extra structure.

A very common pattern in R is to apply functions to vectors (as opposed to using `for` loops). A function that takes a vector as input is called **vectorized**.

Example i

```r
# Create a vector
vect <- c(2, 6, 3, 5.5)

# What is its mean?
# Using for loops
n <- length(vect)
sum <- 0 # Initialize

for (i in 1:n) { # R is 1-indexed!
  sum <- sum + vect[i]
}
```

Example ii

```
mean <- sum/n
mean
```

```
## [1] 4.125
```

```
# In R, use vectorized functions
# whenever possible
mean(vect)
```

```
## [1] 4.125
```

Main object types

Variables

- Variables are ways to assign values (or objects) to names (or symbols).
    - E.g. `vect <- c(1,2,3,4)`
- This allows us to write more robust and flexible code.
- Use meaningful names to make code *human-readable*.
    - Try to use `n.sample` or `sample_size` or `createMatrix`, instead of `n` or `nn`.
    - Have a look at the Tidyverse style guide for `R`: `https://style.tidyverse.org/`
- Descriptive names make it easier to design, debug, and improve your code.

Assignment Operator

- The are several ways of assigning a value to a variable.

```r
# These are all equivalent
x <- 10
10 -> x
x = 10
assign("x", 10)
```

- For readability of the code, the preferred option is `<-`.
  - Although `=` also works, it is usually reserved for function arguments.

Atomic vectors i

- Recall: An atomic **vector** is a sequence of values, all of the same primitive type.

```
x <- c(0, 5, 12, 8)
x
```

```
## [1]  0  5 12  8
```

- The `c` function (for *concatenate*) returns a vector made from all the given arguments.

Atomic vectors ii

```
y <- c(3, 2)
c(x, y)
```

```
## [1]  0  5 12  8  3  2
```

- If elements are not all of same type, R tries to coerce them.

```
c(1, 2.5, "Stat", FALSE)
```

```
## [1] "1"     "2.5"   "Stat"  "FALSE"
```

Atomic vectors iii

- R has a built-in function to create sequences.

```
seq(from = 1, to = 3, by = 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0
```

```
# Equivalently
seq(1, 3, by = 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0
```

Atomic vectors iv

```
# Decreasing sequences
seq(1, 0, by = -0.2)
```

```
## [1] 1.0 0.8 0.6 0.4 0.2 0.0
```

- There is also a shorthand for sequences of consecutive integers:

```
1:5
```

```
## [1] 1 2 3 4 5
```

Atomic vectors v

```
1:(-2)
```

```
## [1]  1  0 -1 -2
```

- Accessing one element of a vector:

```
x[2]
```

```
## [1] 5
```

Atomic vectors vi

- Accessing more than one element:

```
x[c(1, 3)]
```

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

- Accessing all but some elements:

```
x[c(-2, -4)]
```

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

Atomic vectors vii

- Accessing can also be done with a boolean vector:

```
x_large <- x > 7
x_large
```

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

```
x[x_large]
```

```
## [1] 12  8
```

Atomic vectors viii

- Or using the `which` function (returns the *indices* of the elements of a boolean vector that are `TRUE`)

```
which(x_large)
```

```
## [1] 3 4
```

```
x[which(x_large)]
```

```
## [1] 12  8
```

Comparisons and Logical Operators i

- Comparisons are made like most other languages:

```
7 <= 5
```

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

```
7 != 5
```

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

Comparisons and Logical Operators ii

```
# Even works with character values
"abc" < "bca"
```

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

- Recall that = is an assignment operator. Equality is checked with a double equal sign:

```
7 == 5
```

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

Comparisons and Logical Operators iii

- For vectors of length > 1, comparisons are actually done component-wise:

```
y <- rep(10, times = 4) # rep for repeat
y
```

```
## [1] 10 10 10 10
```

```
x < y
```

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

Comparisons and Logical Operators iv

- Because of *recycling*, this is equivalent to:

```
x < 10
```

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

- The basic logical operators are | (or) and & (and). They also work component-wise:

```
(x > 3) & (x < 10)
```

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

Comparisons and Logical Operators v

```
(x < 3) | (x > 10)
```

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

Vector arithmetic

- Arithmetic operators work component-wise:

```
z <- 1:4
x + z
```

```
## [1]  1  7 15 12
```

```
x * z
```

```
## [1]  0 10 36 32
```

```
z / x
```

```
## [1]  Inf 0.40 0.25 0.50
```

Recycling

- Binary operators (arithmetic, comparison, logical) are applied element-wise to vectors.
- R uses the concept of **recycling** when applying these operators to vectors of different lengths:
    - repeat the shorter vector enough times to obtain a new vector of the same length as the longer vector
    - apply the operator to the two longer vectors thus obtained
    - if the length of the longer vector is not a multiple of the length of the shorter vector, R returns a warning.

Arrays and Matrices i

- **Arrays** are tables made from elements of the same type, like atomic vectors.
- You can create arrays from atomic vectors by specifying the dimensions.
    - **Note**: R is column-major, which means it fills the matrix column by column (instead of by row)

```
A_mat <- matrix(1:4, nrow = 2, ncol = 2)
A_mat
```

Arrays and Matrices ii

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

- If you prefer filling by row:

```
B_mat <- matrix(1:4, nrow = 2, ncol = 2, byrow = TRUE)
B_mat
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

Arrays and Matrices iii

- Array arithmetic (`+`, `*`, etc.) is done component-wise:

```
A_mat * B_mat
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    6   16
```

Accessing elements of arrays i

- Accessing elements of arrays can be done by proper indexing of the array itself or by indexing the underlying vector:

```
A_mat[1, 2]
```

```
## [1] 3
```

```
A_mat[3]
```

```
## [1] 3
```

Accessing elements of arrays ii

- You can also select a full row or columns:

```
B_mat[, 2]
```

```
## [1] 2 4
```

Basic matrix operations i

- Some basic operations: `t`, `det`, `%*%`.
- Matrix multiplication requires the dimension of the involved matrices to match.

```
B_mat %*% t(A_mat)
```

```
##      [,1] [,2]
## [1,]    7   10
## [2,]   15   22
```

- R treats vectors as column-vectors or row-vectors, as needed.

Basic matrix operations ii

```
A_mat %*% c(2, 3)
```

```
##      [,1]
## [1,]   11
## [2,]   16
```

```
c(5, 1) %*% A_mat
```

```
##      [,1] [,2]
## [1,]    7   19
```

Basic matrix operations iii

- Matrix inversion is done with `solve`:

```
solve(A_mat)
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

Lists i

- Unlike atomic vectors, **lists** are sequences of values, not necessarily all of the same type.

```
course <- list(3150, "Statistical Computing",
               FALSE, 3.0)
course
```

Lists ii

```
## [[1]]
## [1] 3150
##
## [[2]]
## [1] "Statistical Computing"
##
## [[3]]
## [1] FALSE
##
## [[4]]
## [1] 3
```

Lists iii

- Lists are vectors, and they can be subsetted using `[ ]`.

```
course[4]
```

```
## [[1]]
## [1] 3
```

- **Note**: the above is still a list! To extract the element, use double brackets:

```
course[[4]]
```

Lists iv

```
## [1] 3
```

- Use `c` to add elements to a list (just like atomic vectors).

```
c("STAT", course)
```

```
## [[1]]
## [1] "STAT"
##
## [[2]]
## [1] 3150
##
```

Lists v

```
## [[3]]
## [1] "Statistical Computing"
##
## [[4]]
## [1] FALSE
##
## [[5]]
## [1] 3
```

Names for lists i

- Very often, elements of a list will be given names.

```
names(course) <- c("Number", "Title",
                   "Lab", "Credit_hours")
course
```

```
## $Number
## [1] 3150
##
## $Title
## [1] "Statistical Computing"
```

Names for lists ii

```
##
## $Lab
## [1] FALSE
##
## $Credit_hours
## [1] 3
```

- There is a shortcut for using names with lists:

```
course[["Title"]]
```

```
## [1] "Statistical Computing"
```

Names for lists iii

```
course$Title
```

```
## [1] "Statistical Computing"
```

- Lists can be created with names.

```
other_course <- list(Number = 4150,
                     Title = "Bayesian Statistics",
                     Lab = FALSE, Credit_hours = 3.0)
other_course
```

Names for lists iv

```
## $Number
## [1] 4150
##
## $Title
## [1] "Bayesian Statistics"
##
## $Lab
## [1] FALSE
##
## $Credit_hours
## [1] 3
```

Names for lists v

- A named element can be added to a list.

```
course$Dept <- "STAT"
course
```

```
## $Number
## [1] 3150
##
## $Title
## [1] "Statistical Computing"
##
## $Lab
```

Names for lists vi

```
## [1] FALSE
##
## $Credit_hours
## [1] 3
##
## $Dept
## [1] "STAT"
```

```
other_course[["Dept"]] <- "STAT"
```

Data Frames i

- A **data frame** is a list of vectors that are all of the same length. Importantly, the vectors *can be of different types*.
- Data frames are how R models datasets:
  - Columns are variables,
  - Rows are units or subjects.

```
courses <- data.frame(Dept = c("STAT", "STAT"),
                      Number = c(3150, 4150),
                      Title =c("Statistical Computing",
                               "Bayesian Statistics"),
                      Lab = c(FALSE, FALSE),
                      Credit_hours = c(3.0, 3.0))
```

Data Frames ii

```
courses
```

```
## Dept Number Title Lab Credit_hours
## 1 STAT 3150 Statistical Computing FALSE 3
## 2 STAT 4150 Bayesian Statistics FALSE 3
```

- Elements of data frames can be accessed, like matrices, by indices or names.

```
courses[2, "Title"]
```

```
## [1] "Bayesian Statistics"
```

Data Frames iii

- The shortcut $ works with the columns of data frames:

```
courses$Lab
```

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

Control Structures

Conditional Statements i

- The function **if** is used to control which of two blocks of code are executed.
- The typical syntax is:

```
if (condition) {
  # Block of code to be executed
  # when condition is TRUE
} else {
  # Another block of code to be executed
  # when condition is FALSE
}
```

Conditional Statements ii

- Braces are not necessary when a block contains only one line of code, but it is good practice to use the above syntax.
- The `else` statement is not required.

Example

```r
# Sample 1 value from a standard normal
x <- rnorm(1)
if (x < 0) {
  message("The observation x is negative.")
} else {
  message("The observation x is positive.")
}
```

```
## The observation x is negative.
```

Conditional Statements (cont'd) i

- It is also possible to have more than one else statement:

```
x <- rnorm(10)
loc_measure <- "mid_point"
```

Conditional Statements (cont'd) ii

```
if (loc_measure == "mean") {
  mean(x)
} else if (loc_measure == "median") {
  median(x)
} else if (loc_measure == "mid_point") {
  0.5*(min(x) + max(x))
} else {
  stop(paste("You have to choose between mean,",
             "median and mid_point."))
}
```

Conditional Statements (cont'd) iii

```
## [1] 0.4846854
```

- The above is referred to as a *nested if* structure.
- The `switch` function can also be used in the above setting:

Conditional Statements (cont'd) iv

```
loc_measure <- "truncated_mean"
switch(loc_measure,
       "mean" = mean(x),
       "median" = median(x),
       "mid_point" = 0.5*(min(x) + max(x)),
       stop(paste("You have to choose between mean,\n",
            "median and mid_point."), call. = FALSE)
       )
```

```
## Error: You have to choose between mean,
##  median and mid_point.
```

`for` Loop i

- The `for` statement specifies that a certain operation should be repeated a *fixed* number of times.
- The syntax is:

```
for (element in vector) {
  # Block of code to be repeated
  # once for each element of vector
}
```

Example

```
for (k in 3:0) {
  message(k)
  if (k == 0) message("Blast off!")
}
```

```
## 3

## 2

## 1

## 0

## Blast off!
```

Exercise

Approximate a geometric sum using a finite number N of terms:

$$S = \sum_{k=0}^{\infty} \left(\frac{1}{2}\right)^k \approx \sum_{k=0}^{N-1} \left(\frac{1}{2}\right)^k .$$

Solution i

```r
N <- 10
approx <- 0
for (k in 0:(N - 1)) {
  # update current approx by adding next term
  approx <- approx + 2^{-k}
}
approx
```

```
## [1] 1.998047
```

- **Note**: it is more efficient to use vectorized functions:

Solution ii

```
sum(0.5^(0:(N-1)))
```

```
## [1] 1.998047
```

`while` Loop

- The *while* loop repeats an expression for as long as a condition holds.
- The syntax is:

```
while (condition) {
  # Block of code to be repeated
  # as long as condition is TRUE
}
```

Example i

```
num_flips <- 0
flip <- "tails"

while (flip == "tails") {
  # Flip a coin
  flip <- sample(c("tails", "heads"), size = 1)
  num_flips <- num_flips + 1
}
```

Example ii

```
# How many flips?
num_flips
```

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

Example i

- What if we want to approximate the geometric sum within $\epsilon = 10^{-7}$ of the true value of S?

```
approx <- 0
current_err <- 2
k <- 0
while(current_err > 10^(-7)) {
  approx <- approx + 2^(-k)
  current_err <- 2 - approx
  k <- k + 1
}
```

Example ii

```
# How many terms?
k
```

```
## [1] 25
```

Creating functions i

- Creating new functions is an important part of programming.
- This is done with the function `function` and through assignment.

```
new_function <- function(arg1, arg2 = def_val) {
  # Block of code to be executed using the arguments
  return(value)
}
```

- This creates a function named `new_function` that can then be used like any other R function.

Creating functions ii

- The function has two arguments:
  - `arg1` is required
  - `arg2` has the default value `def_val`
- The function will return the output of the last statement, unless it hits `return` (after which it exits).

Example i

- Consider the following piecewise linear function:

$$f(x) = \begin{cases} 1 & \text{if } x < -1 \text{ or } x > 1, \\ -x & \text{if } -1 \leq x < 0, \\ x & \text{if } 0 \leq x \leq 1. \end{cases}$$

- In R, this can be defined as follows:

Example ii

```r
fun <- function(x) {
  if ((x >= -1) & (x < 0)) {
    value <- -x
  } else if ((x >= 0) & (x <= 1)) {
    value <- x
  } else value <- 1

  return(value)
}
```

Example iii

```
c(fun(-3), fun(-0.3), fun(0.4), fun(1.5))
```

```
## [1] 1.0 0.3 0.4 1.0
```

- Equivalently, we can define

Example iv

```
fun_cleaner <- function(x) {
  if ((x >= -1) & (x < 0)) {
    return(-x)
  }
  if ((x >= 0) & (x <= 1)) {
    return(x)
  }
  return(1)
}
```

Example v

```
c(fun_cleaner(-3), fun_cleaner(-0.3),
  fun_cleaner(0.4), fun_cleaner(1.5))
```

```
## [1] 1.0 0.3 0.4 1.0
```

Exercise

- Implement the following, more general, function, where $a > 0$ is arbitrary:

$$f_a(x) = \begin{cases} a & \text{if } x < a, \text{ or } x > a, \\ -x & \text{if } -a \leq x < 0, \\ x & \text{if } 0 \leq x \leq a, \end{cases}$$

- *Hint*: Use a second argument to the function.