Review of Fundamental Concepts

Chapter 1

Stats 102A: Introduction to Computational Statistics with R

UCLA

Acknowledgements: Miles Chen and Michael Tsiang

Section 1

Basic Data Structures

Vectors

The most important family of objects in R is **vectors**.

There are two types of vectors: atomic vectors and generic vectors (also called lists).

Question: What is the main difference between an atomic vector and a list?

Atomic Vectors

The most fundamental object in R is an **atomic vector** (or vector), which is an ordered collection of values.

Atomic vectors have six basic types (though we will only work with the first four):

"logical", "integer", "double", "character", "complex" and "raw".

All elements of an atomic vector must be of the same type.

Doubles and Integers

The conversion between integer and double values is often done automatically in R, so the distinction is typically not needed (they are both numeric types).

However, to be complete, doubles are numeric values stored with floating point precision, while integers are stored as exact integer values.

Integers are indicated by an `L` after the number, and vectors of integers can be created using the colon (`:`) operator.

Doubles and Integers

```
typeof(c(1,2,3))
```

```
## [1] "double"
```

```
is.double(pi)
```

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

```
typeof(1L)
```

```
## [1] "integer"
```

```
is.integer(1:3)
```

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

Lists

A **list** (or generic vector) is an ordered collection of objects. Lists are the most flexible objects in R, as each component in a list can be *any* other object, including other lists.

The most common objects built from vectors are summarized in the table below.

Dimension	Homogeneous	Heterogeneous
1-dim	Atomic vector	List (generic vector)
2-dim	Matrix	Data frame
n-dim	Array	

Attributes

Every vector (atomic or generic) can also have **attributes**, which is a named list of arbitrary metadata.

Two attributes are particularly important: The **dimension** attribute turns vectors into matrices and arrays, and the **class** attribute develops the **S3** object system (which we will cover later in the course).

Question: How do you get and set attributes of an object?

Question: How is a matrix different from a data frame? How is each one internally stored in R?

Question: What is a factor? How is a factor internally stored in R?

Attributes

The **attr()** function can be used to get or set a single attribute of an object.

The **attributes()** function can be used to access the entire list of attributes.

```
a <- 1:3
attr(a, "x") <- "This is a attribute"
attr(a, "y") <- 2:8
str(attributes(a))
```

```
## List of 2
##  $ x: chr "This is a attribute"
##  $ y: int [1:7] 2 3 4 5 6 7 8
```

Matrices

A **matrix** in R is a vector with a dimension attribute of length 2.

```
M <- 1:12
attr(M,"dim") <- c(3, 4)
M
```

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

```
attributes(M)
```

```
## $dim
## [1] 3 4
```

```
class(M)
```

```
## [1] "matrix" "array"
```

Arrays

An **array** in R is a vector with a dimension attribute of length more than 2.

```
A <- 1:12
attr(A,"dim") <- c(2,3,2)
A
```

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

Arrays can also be created using `array()`.

Data Frame

A **data frame** in R is internally stored as a list of equal length vectors with a class attribute called `data.frame`.

```
typeof(trees)
```

```
## [1] "list"
```

```
attributes(trees)
```

```
## $names
## [1] "Girth"  "Height" "Volume"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
## [16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
## [31] 31
```

Factors

A **factor** is a vector used to represent categorical values. It is internally stored as an integer vector with levels and class attributes.

```
gender <- c("M","F","F","X","M","F")
gender_fac <- factor(gender)
gender_fac
```

```
## [1] M F F X M F
## Levels: F M X
```

```
levels(gender_fac)
```

```
## [1] "F" "M" "X"
```

Coercion

To illustrate the idea of coercion, we create the following vectors.

```
l <- c(TRUE, FALSE, TRUE)
i <- 1L
d <- c(5, 6, 7)
ch <- c("a", "b", "c")
```

When values of different types are concatenated into a single vector, the values are **coerced** into a single type.

Question: What is the output for the following commands?

```
typeof(c(l, i, d))
typeof(c(l, d, ch))
```

Coercion

```
typeof(c(l, i, d))
```

```
## [1] "double"
```

```
typeof(c(l, d, ch))
```

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

Explicit Coercion

The as functions can be used to explicitly coerce objects, if possible.

```
trials <- c(FALSE, FALSE, TRUE)
as.numeric(trials)
```

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

```
as.character(trials)
```

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

```
as.logical(c(2, "TRUE", 0))
```

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

```
as.numeric("cat")
```

```
## Warning: NAs introduced by coercion
```

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

Implicit Coercion

Coercion often happens automatically. Most mathematical functions (+, `log()`, `abs()`, etc.) will coerce to a double or integer, and most logical operations (`&`, `|`, `any()`, etc.) will coerce to a logical.

```
# Total number of TRUEs
sum(trials)
```

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

```
# Proportion that are TRUE
mean(trials)
```

```
## [1] 0.3333333
```

Special Values

Question: What is the difference between `NA`, `NULL`, and `NaN`?

Special Values

Question: What is the difference between `NA`, `NULL`, and `NaN`?

- **`NA`** is used to represent missing or unknown values.
- **`NULL`** is used to represent an empty or nonexistent value.
- **`NaN`** is used to represent indeterminate forms in mathematics (such as `0/0` or `log(-1)`).

Vector Arithmetic

Arithmetic can be done on numeric vectors using the usual arithmetic operations. The operations are **vectorized**, i.e., they are applied elementwise (to each individual element).

```
x <- c(1, 3, 5)
y <- c(2, 4, 3)
x + y
```

```
## [1] 3 7 8
```

```
x * y
```

```
## [1]  2 12 15
```

Vector Recycling

When applying arithmetic operations to two vectors of different lengths, R will automatically **recycle**, or repeat, the shorter vector until it is long enough to match the longer vector.

Question: What is the output of the following commands?

```
c(1, 3, 5) + c(5, 7, 0, 2, 9, 11)
c(1, 3, 5) + c(5, 7, 0, 2, 9)
```

Question: When will R throw a warning when recycling?

Vector Recycling

```
c(1, 3, 5) + c(5, 7, 0, 2, 9, 11)
```

```
## [1]  6 10  5  3 12 16
```

```
c(1, 3, 5) + c(5, 7, 0, 2, 9)
```

```
## Warning in c(1, 3, 5) + c(5, 7, 0, 2, 9): longer
## object length is not a multiple of shorter object
## length
```

```
## [1]  6 10  5  3 12
```

Section 2

Subsetting

Subsetting

Extracting values or components from objects is called **subsetting**.

Question: How would you subset the `F` values from the `gender_fac` vector?

```
gender_fac
```

```
## [1] M F F X M F
## Levels: F M X
```

Numeric Indices

Positive indices:

```
gender_fac[c(2, 3, 6)]
```

```
## [1] F F F
## Levels: F M X
```

Negative indices:

```
gender_fac[-c(1, 4, 5)]
```

```
## [1] F F F
## Levels: F M X
```

Logical Indices

A more general (programmatic) way would be to use logical indexing. This approach does not depend on the exact order of the values.

```
gender_fac[gender_fac == "F"]
```

```
## [1] F F F
## Levels: F M X
```

```
gender_fac[gender_fac != "M" & gender_fac != "X"]
```

```
## [1] F F F
## Levels: F M X
```

Question: What does `gender_fac[]` return?

Empty Indices

```
gender_fac[]
```

```
## [1] M F F X M F
## Levels: F M X
```

Empty indices return everything.

Subsetting from Lists

Both single `[` and double square brackets `[[` can be used for subsetting.

```
L <- list("Vector" = 1:6, "Matrix" = M,
          "Factor" = gender_fac)
```

Question: What is the difference between `L[3]` an `L[[3]]`?

Subsetting from Lists

The single square bracket returns a list object. The double square bracket returns only the component inside the list.

```
L[3]
```

```
## $Factor
## [1] M F F X M F
## Levels: F M X
```

```
L[[3]]
```

```
## [1] M F F X M F
## Levels: F M X
```

Question: How else (not using numeric indices) can you subset the matrix component of `L`?

Character Indices

When components are named, the names can be used as character indices.

```
L["Matrix"]
```

```
## $Matrix
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
L[["Matrix"]]
```

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

The $ Operator

The $ operator can also be used to subset named components of a list object.

```
L$Matrix
```

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

```
L$Mat
```

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

The $ operator allows for partial name matching, as long as the desired component is unambiguous.

Subsetting from Two-Dimensional Objects

Question: How do you subset values from two-dimensional objects, such as matrices or data frames?

```
M
```

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

```
head(trees)
```

```
##   Girth Height Volume
## 1   8.3     70   10.3
## 2   8.6     65   10.3
## 3   8.8     63   10.2
## 4  10.5     72   16.4
## 5  10.7     81   18.8
## 6  10.8     83   19.7
```

Subsetting from Two-Dimensional Objects

Using single square brackets for two dimensions requires two indices: `[row index,column index]`.

```
M[2, 4]
```

```
## [1] 11
```

```
trees[4, ]
```

```
##   Girth Height Volume
## 4  10.5     72   16.4
```

```
trees[-(1:10), "Girth"]
```

```
##  [1] 11.3 11.4 11.4 11.7 12.0 12.9 12.9 13.3 13.7
## [10] 13.8 14.0 14.2 14.5 16.0 16.3 17.3 17.5 17.9
## [19] 18.0 18.0 20.6
```

drop = FALSE

The **drop = FALSE** argument in single square brackets can be used to retain object structure when subsetting.

```
M[2, 4, drop=FALSE]
```

```
##      [,1]
## [1,]   11
```

```
trees[-(1:28), "Girth", drop=FALSE]
```

```
##    Girth
## 29  18.0
## 30  18.0
## 31  20.6
```

```
trees[-(1:28), drop=FALSE]
```

```
## Warning in `[.data.frame`(trees, -(1:28), drop =
## FALSE): 'drop' argument will be ignored
```

Data Frames Are Lists

Since data frames are stored as lists, the double square bracket and $ operator can also be used for data frames.

```
trees[["Volume"]]
```

```
##  [1] 10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6
## [10] 19.9 24.2 21.0 21.4 21.3 19.1 22.2 33.8 27.4
## [19] 25.7 24.9 34.5 31.7 36.3 38.3 42.6 55.4 55.7
## [28] 58.3 51.5 51.0 77.0
```

```
trees$Height[trees$Girth > 9.5]
```

```
##  [1] 72 81 83 66 75 80 75 79 76 76 69 75 74 85 86
## [16] 71 64 78 80 74 72 77 81 82 80 80 80 87
```

Section 3

Control Statements

Conditional Execution: The `if()` Statement

The `if()` statement is used for conditional execution, meaning the commands inside an `if` statement are executed only if specified conditions are met.

The syntax of an `if()` statement is given by:

```
if(condition){
   # Commands when TRUE
}
```

The `condition` must be a logical expression that produces a single logical value, either `TRUE` or `FALSE`.

Conditional Execution: The `if-else` Statement

If there is an alternative set of commands to execute when `condition` is `FALSE`, then an **`else`** statement is added to the `if` statement.

The syntax for an **`if-else`** statement is given by:

```
if(condition){
   # Commands when TRUE
} else{
   # Commands when FALSE
}
```

Conditional Execution: The `ifelse()` Function

A vectorized version of the `if-else` statement is the **`ifelse()`** function.

The `ifelse()` function has the form

```
ifelse(condition_vector, x, y)
```

and returns a vector of the same length as `condition_vector`, with elements `x` if `condition[i]` is `TRUE` and `y` if `condition[i]` is `FALSE`.

```
ifelse(1:8 %% 2 == 0, "even", "odd")
```

```
## [1] "odd"  "even" "odd"  "even" "odd"  "even"
## [7] "odd"  "even"
```

Repeated Execution: The `for()` Loop

The **for loop** is a common coding procedure in most (if not all) programming languages that repeats a set of commands a fixed number of times. The `for()` statement is used to create for loops in R.

The syntax of a `for()` loop is given by:

```
for(name in vector){
  # Commands go here
}
```

The `for()` statement performs one iteration of the loop for each entry in `vector`, with the `name` variable being assigned to the values in those entries:

The first iteration assigns `name <- vector[1]`, the second iteration assigns `name <- vector[2]`, etc.

Example: The Fibonacci Sequence

The **Fibonacci sequence** is a famous and well-studied sequence in mathematics. The first two terms in the sequence are 1 and 1, and each subsequent number is the sum of the previous two terms. For example, the third term is $1 + 1 = 2$, the fourth term is $1 + 2 = 3$, the fifth term is $2 + 3 = 5$, etc.

Example: The Fibonacci Sequence

The **Fibonacci sequence** is a famous and well-studied sequence in mathematics. The first two terms in the sequence are 1 and 1, and each subsequent number is the sum of the previous two terms. For example, the third term is $1 + 1 = 2$, the fourth term is $1 + 2 = 3$, the fifth term is $2 + 3 = 5$, etc.

The `for()` loop below computes the first 12 Fibonacci numbers.

```
fib <- numeric(12)
fib[1:2] <- c(1, 1)
for(i in 3:12){
  fib[i] <- fib[i-2] + fib[i-1]
}
fib
```

```
##  [1]   1   1   2   3   5   8  13  21  34  55  89
## [12] 144
```

Repeated Execution: The `while()` Loop

The `for()` loop repeats a set of commands a fixed number of times. In some scenarios, the number of times to repeat the commands is not known in advance, so a different type of loop needs to be used.

The **`while()`** statement creates a loop that repeats a set of commands for as long as a certain condition holds.

The syntax of a `while()` loop is given by:

```
while(condition){
  # Commands go here
}
```

Question: How would you rewrite the Fibonacci `for()` loop to produce all the Fibonacci numbers less than 1000?

Repeated Execution: The `repeat` Loop

The **`repeat`** statement creates a loop that executes a set of commands repeatedly without a built-in `condition` to exit the loop. The loop will repeat indefinitely until a **`break`** statement is executed.

The syntax of a `repeat` loop is given by:

```
repeat{
   # Commands go here
   if(condition){break}
}
```

Question: How would you rewrite the Fibonacci `while()` loop using a `repeat` statement?

Section 4

Functions

Functions

Functions are a special type of object in R.

A function in R takes in certain input objects called **arguments** and returns an output by executing a set of commands.

The basic syntax to create your own function is:

```
function_name <- function(arg1, arg2, ...){
  # The body of the function goes here.
}
```

Once a function is created, a call to the function is then implemented as:

```
function_name(expr1, expr2, ...)
```

Argument Matching

R function arguments can be matched positionally or by name. If we know the order of the arguments, we do not need to use the name in order to specify the arguments. However, it is typically better to use named arguments for clarity.

```
matrix(1:10, 2, 5)
matrix(data=1:10, ncol=5, nrow=2)
```

```
pnorm(1.96, , ,FALSE)
pnorm(lower.tail = FALSE, q = 1.96)
```

Notice that blank arguments do not erase default values.

Writing a Function

The sample variance of data values $x = (x_1, x_2, \ldots, x_n)$ is

$$\text{Var}(x) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2,$$

where $\bar{x} = \sum_{i=1}^{n} x_i / n$ is the sample mean.

Question: How would you write a function to compute the variance of a numeric vector of data (not using the built-in `var()` function)?

Writing a Function

An example of a function that computes variance:

```
var_fn <- function(x){
  n <- length(x)
  xbar <- mean(x)
  squared_devs <- (x - xbar)^2
  output <- sum(squared_devs) / (n - 1)
  output
}
```

Writing a Function (Cont.)

We can verify that the function computes the correct value:

```
lost_nums <- c(4, 8, 15, 16, 23, 42)
var_fn(lost_nums)
```

```
## [1] 182
```

```
var(lost_nums)
```

```
## [1] 182
```

Lazy evaluation

In R, function arguments are lazily evaluated: they're only evaluated if accessed.

```
h01 <- function(x) {
  10
}
h01(stop("This is an error!"))
```

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

The ... Argument

To write functions with more general or flexible functionality, it is often helpful to allow one function to pass arguments to another.

The `...` argument is used to pass optional arguments to functions used inside the main function.

```
cor_plot <- function(x, y, scatter=TRUE,...){
  if(scatter){
    plot(x, y, ...)
  }
  cor(x, y)
}
```

In this example, the `...` enables the `cor_plot()` function to pass arguments to the `plot()` function.

```
cor_plot(x, y, col="dark red", pch = 16)
```

Anonymous function

While you almost always create a function and then bind it to a name, the binding step is not compulsory. If you choose not to give a function a name, you get an anonymous function.

```
lapply(mtcars, function(x) length(unique(x)))
```

```
funs <- list(
  half = function(x) x / 2,
  double = function(x) x * 2
)
funs$double(10)
```

```
## [1] 20
```

Function forms

While everything that happens in R is a result of a function call, not all calls look the same. Function calls come in four varieties:

– **prefix:** the function name comes before its arguments, like foofy(a, b, c).

– **infix:** the function name comes in between its arguments, like x + y.

– **replacement:** functions replace values by assignment, like names(df) <- c("a", "b", "c").

– **special:** functions like [[, if, and for.

Prefix form

The prefix form is the most common form in R code, and indeed in the majority of programming languages. Prefix calls in R are a little special because you can specify arguments in three ways:

– By name

– By position

– Using partial matching

For example,

```
k1 <- function(abcd, bcd1, bcd2) {
  list(a = abcd, b1 = bcd1, b2 = bcd2)
}
```

Prefix form

The prefix form is the most common form in R code, and indeed in the majority of programming languages. Prefix calls in R are a little special because you can specify arguments in three ways:

– By name

```
str(k1(bcd1 = 2, bcd2 = 3, abcd = 1))
```

```
## List of 3
##  $ a : num 1
##  $ b1: num 2
##  $ b2: num 3
```

– By position

– Using partial matching

Prefix form

The prefix form is the most common form in R code, and indeed in the majority of programming languages. Prefix calls in R are a little special because you can specify arguments in three ways:

– By name

– By position

```
str(k1(1, 2, 3))
```

```
## List of 3
##  $ a : num 1
##  $ b1: num 2
##  $ b2: num 3
```

– Using partial matching

Prefix form

The prefix form is the most common form in R code, and indeed in the majority of programming languages. Prefix calls in R are a little special because you can specify arguments in three ways:

- By name
- By position
- Using partial matching

```
str(k1(2, 3, a = 1))
```

```
## List of 3
##  $ a : num 1
##  $ b1: num 2
##  $ b2: num 3
```

```
str(k1(2, 3, b = 1))
```

Infix functions

Infix functions get their name from the fact the function name comes in between its arguments, and hence have two arguments. R comes with a number of built-in infix operators: :, ::, :::, $, @, ^, *, /, +, -, >, >=, <, <=, ==, !=, !, &, &&, |, ||, ~, <-, and «-. You can also create your own infix functions that start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
"new " %+% "string"
```

```
## [1] "new string"
```

Replacement functions

Replacement functions act like they modify their arguments in place, and have the special name xxx<-.

They must have arguments named x and value, and must return the modified object.

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
```

```
x <- 1:10
second(x) <- 5L
typeof(x)
```

```
## [1] "integer"
```

Error Handling

What if the input vector contains `NA` values?

```
incomplete_nums <- c(4, 8, NA, 16, 23, NA)
var_fn(incomplete_nums)
```

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

It would be helpful if our function could throw an error or a warning message if it cannot compute the variance properly.

Question: How would you modify the `var_fn()` function to check for `NA` values?

Error Handling: The stop() Function

The **stop()** function stops the execution of the current expression and throws an error message.

```
var_fn2 <- function(x){
  if(sum(is.na(x)) > 0){
    stop("The input has NA values!")
  }
  n <- length(x)
  xbar <- mean(x)
  squared_devs <- (x - xbar) ^ 2
  output <- sum(squared_devs) / (n - 1)
  output
}
var_fn2(incomplete_nums)
```

```
## Error in var_fn2(incomplete_nums): The input has NA values!
```

Error Handling: The warning() Function

The **warning()** function throws a warning message but does not stop the execution of the current expression.

```
var_fn3 <- function(x){
  if(sum(is.na(x)) > 0){
    warning("The input has NA values!")
  }
  n <- length(x)
  xbar <- mean(x)
  squared_devs <- (x - xbar) ^ 2
  output <- sum(squared_devs) / (n - 1)
  output
}
var_fn3(incomplete_nums)
```

```
## Warning in var_fn3(incomplete_nums): The input
## has NA values!
```

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

Error Handling

Think about other things that can go wrong with this function, or what other functionality you might want:

- What if the input object is not numeric?
- What if the input object is not an atomic vector?
- What if you wanted to first check for `NA` values, throw a warning if necessary, then remove the `NA` values before computing the variance?

Example: The Sieve of Eratosthenes

The oldest known algorithm for finding all prime numbers up to any given number (say n) is called the **sieve of Eratosthenes**.

The main idea is as follows:

1. Start with a list of consecutive numbers from 2 to n.
2. Set $p = 2$, the smallest prime.
3. Eliminate all multiples of p from the list that are larger than p, i.e., $2p, 3p, 4p$, etc.
4. Set p to be the next number in the list that has not been eliminated (i.e., the next prime). Repeat from Step 3.
5. The algorithm stops when there are no numbers left in the list larger than p. The remaining list contains all primes up to n.

A helpful image:
https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#/media/File:

Example: The Sieve of Eratosthenes

An example of a `for()` loop that computes the sieve of Eratosthenes for $n = 100$.

```
soe <- function(n){
  if(!is.numeric(n)){
      stop("The input is not numeric!")
  }
  sieve <- seq(2, length.out=n-1)
  for(p in seq(2, length.out=n-1)){
    if(any(p == sieve)){
      p_multiples <- (sieve %% p) == 0
      sieve <- c(sieve[!p_multiples], p)
    }
  }
  sieve
}
```

Example: The Sieve of Eratosthenes (Cont.)

```
n <- 100
soe(n)
```

```
##  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47
## [16] 53 59 61 67 71 73 79 83 89 97
```