R Programming

# Week 1

## History and Overview of R

### What is R?

* R is a dialect of the S language

### What is S?

* A language developed at Bell Labs
* Was initiated as an internal statisitcs analysis environment, originally implemented as Fortran libraries
* Was later rewritten in C
* People would enter the language in an interactive environemt, without knowing too much programming or knowledge of the language.

## R Input and Evaluation

### Entering Input

* Things typed into the R prompt are called **expressions**
* > x <- 1 is an expression
* In this case, x is an R **numeric** object, a numeric vector really where the first element is 1
* msg <- "hello"
* The above expression assigns a character vector to msg. The first element of this **character** vector is “hello”
* Comments in R are created with the # symbol

## Evaluation

* When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be autoprinted
* When you just type an object’s name and hit enter, R will by default **autoprint** the value of that object. IT’s the same as calling the print() function on that object
* The str() function compactly displays the structure of any R object
* Use the ls() function to print out all the objects in the working environment

### Printing

* The “:” operator creates integer sequences
* > x <- 1:20
* > x
* [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

## Help

* help()
* vignette()

## Objects and Attributes

### Objects

* R has 5 basic or **atomic** classes of objects
  + character
  + numeric (real numbers)
  + integer
  + complex
  + logical
* The most basic object is a **vector**
  + A vector can contain objects of any class, but it must be homogenous. I.e., they must all be of the same class
  + A **list** which is represented as a vector can contain different classes allowing it to be
  + Vectors can be created with the vector() function

### Numbers

* They are usually treated as numeric objects (double precision reals)
* IF you explicitly want an integer, you need to suffix a number with L
* Entering 1 gives you a numeric, entering 1L gives you an integer
* Inf represents infinity
* NaN represents undefined values

### Attributes

* R objects can have attributes
  + names, dimnames
  + dimensions (arrays, matrices)
  + class
  + length
  + other user defined attributes
* Attributes of an object can be obtained with the attributes() function

## Vectors and Lists

## Vectors come in two different flavors: atomic vectors and lists. An atomic vector contains exactly one data type, whereas a list may contain multiple data types.

## We'll explore atomic vectors further before we get to lists.

### Creating Vectors

* The c() function is used to create vectors of objects.
  + Think of it as standing for concatenate
* > x <- c(1,2,3)
* > x
* [1] 1 2 3
* We can also use the vector() function to create a vector as so
* > x <- vector("numeric", 3)
* > x
* [1] 0 0 0
* This just creates a numeric vector of length 3 with the values al set to 0.
* This way of creating a vector is far less dynamic

### Mixing Objects

* What happens when you try to make a vector of 2 different types of objects?
* > x <- c(1, "A")
* > x
* [1] "1" "A"
* Something called coercion occurs where every element in the vector is made to be of the same class. So in our example, the numeric 1 got converted to a character

### Explicit Coercion

* Objects can be explicitly coerced from one class to another using the function as.\* where \* is the class name that you want to coerce an object into.
* > x <- 0:6
* > class(x)
* [1] "integer"
* > as.numeric(x)
* [1] 0 1 2 3 4 5 6
* > as.logical(x)
* [1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
* > as.character(x)
* [1] "0" "1" "2" "3" "4" "5" "6"
* Note that coercion doesn’t always work!
* If you try to convert c(“a”, “b”, “c”) to numeric let’s say, you get a vector of NAs

### Lists

* Can contain elements of different classes
* Are created using the list() function
* > x <- list("a", TRUE, 1)
* > x
* [[1]]
* [1] "a"
* [[2]]
* [1] TRUE
* [[3]]
* [1] 1
* Noticed how the elements are indexed by double brackets and not single brackets
  + This implies that we need to do x[[1]] to get “a” itself.
  + Doing x[1] gets us a list with “a” as the only element of that list

## Matrices

### Basics and Creation

* Are a special type of vector in R
* Thery\re not a separate class of attributres, but they’re basically vectors with a special attribute called **dimension**
* **Dimension** is itself a vector of length 2.
  + The first element is the number of rows
  + The second element is the number of columns
* Matrices are created with the matrix() function where the number of rows and columns must be specified. Having a vector of data is optional. IF no data is provided, the matrix has NA values at each index.
* > m <- matrix(nrow = 2, ncol=3)
* > m
* [,1] [,2] [,3]
* [1,] NA NA NA
* [2,] NA NA NA
* > dim(m)
* [1] 2 3
* > attributes(m)
* $dim
* [1] 2 3

### Continued

* Matrices are constructed column-wise, so entries can be thought of starting in the upper left corner and running down the columns
* > m <- matrix(1:6, nrow=2, ncol=3)
* > m
* [,1] [,2] [,3]
* [1,] 1 3 5
* [2,] 2 4 6
* R takes 1 and 2 to make the first column, 3 and 4 to make up the second column, and 5 and 6 to make up the third column

### Matrix Directly from Vector

> m <- 1:10

> m

[1] 1 2 3 4 5 6 7 8 9 10

> dim(m)

NULL

> dim(m) <- c(2,5)

> m

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

[1,] 1 3 5 7 9

[2,] 2 4 6 8 10

### Column Binding and Row Binding

* Suppose we have 2 vectors x and y, if we cbind x and y, we get a matrix where the first column is 1 to 3 and the second column is 10 to 12.
* > x <- 1:3
* > y <- 10:12
* > cbind(x,y)
* x y
* [1,] 1 10
* [2,] 2 11
* [3,] 3 12
* > rbind(x,y)
* [,1] [,2] [,3]
* x 1 2 3
* y 10 11 12

## Factors

### Basics

* A factor is a special type of vector which is used to represent categorical data.
* There are 2 types of factors: **ordered** and **unordered**
* One can think of a factor as an integer vector where each integer has a label
* **Important**
  + Factors are treated specially by modelling functions such as lm() and glm()
  + Using factors with labels is better than using integers because factors are self-describing

### Creation

* Factors are created using the factor() function
* The input into this function is a character vector
* > x <- factor(c("yes", "no", "yes", "no", "no"))
* > x
* [1] yes no yes no no
* Levels: no yes
* Notice the levels attribute
  + There are only two levels in the factor that we created
* We can call table() function on x to get a frequency count of how many of each level there is in our factor
* > x <- factor(c("yes", "no", "yes", "no", "no"))
* > x
* [1] yes no yes no no
* Levels: no yes
* > table(x)
* x
* no yes
* 3 2

### Unclass

* The unclass() function strips out the class for a vector
* If we call unclass() on x, it will bring it down to an integer vector to see what number represents what level internally by R
* > unclass(x)
* [1] 2 1 2 1 1
* attr(,"levels")
* [1] "no" "yes"

### Order of Levels

* The order of the levels can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level
* > x <- factor(c("yes", "no", "yes", "no", "no"), level = c("yes", "no"))
* > x
* [1] yes no yes no no
* Levels: yes no
* If you don’t specify the levels argument, R takes the levels in alphabetical order so that would be “no” then “yes” in this example

## Missing Values

### Basics

* Are denoted either by NA or Nan.
* Nan is used for undefined mathematical operations whereas NA is used for everything else

### Functions

* is.na() is used to test objects if they are NA
* is.nan() is used to test objects if they are NaN
* Both functions return a logical vector
* NA values have a class also
  + So there is an integer NA
  + A character NA
  + etc.
* A NaN value **IS** considered to be NA but **NOT VICE VERSA**

### Examples

> x <- c(1, 2, NA, 10, 3)

> is.na(x)

[1] FALSE FALSE TRUE FALSE FALSE

> is.nan(x)

[1] FALSE FALSE FALSE FALSE FALSE

* So this is a numeric vector, and therefore the class of the NA value in this vector will be numeric
* Notice how is.nan() returns all false for this vector since there are no NaN values in this vector

> x <- c(1, 2, NaN, NA, 4)

> is.na(x)

[1] FALSE FALSE TRUE TRUE FALSE

> is.nan(x)

[1] FALSE FALSE TRUE FALSE FALSE

* Notice how is.na() returns true for both the NaN and the NA values whereas is.nan() returns true only for the NaN value.

## Data Frames

### Basics

* Used to store tabular data
* Represented as a special type of list where every element of the list has to have the same length
* Each element of the list can be thought of as a column
* The length of each element of the list is the number of rows
* Unlike matrices, data frames can store different classes of objects in each column, just like lists; matrices must have each column be of the same class
* Data frames have a special attribute called row.names
  + Every row of a data frame has a name
  + This can be useful for annotating the data
* Data frames are usually created using read.table() or read.csv()
* Can be converted into a matrix using data.matrix()
  + Mind you this will employ coercion when needed so you may not get exactly what you think/want

### Creating From Scratch

* Use the data.frame() function to create a data frame using vectors.
* > x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
* > x
* foo bar
* 1 1 TRUE
* 2 2 TRUE
* 3 3 FALSE
* 4 4 FALSE
* > nrow(x)
* [1] 4
* > ncol(x)
* [1] 2
* > class(x)
* [1] "data.frame"
* We can specify the row.names argument to set names for every row
* > x <- data.frame(foo = 1:4, bar = c(T, T, F, F), row.names=c("a","b","c","d"))
* > x
* foo bar
* a 1 TRUE
* b 2 TRUE
* c 3 FALSE
* d 4 FALSE

## Names Attribute

### Basics

* R objects can also have names
* This is useful for making readable code and self-described objects

### Setting Names of Vector

* We can set the names of a vector with the names() function
* > x <- 1:3
* > names(x)
* NULL
* > names(x) <- c("foo", "bar", "norf")
* > x
* foo bar norf
* 1 2 3
* > names (x)
* [1] "foo" "bar" "norf"
* So notice how initially the names are null

### Setting Names of Lists

* IT turns out that lists can also have names
* We can create a list with names by specifying the names in the list() function
* > x <- list(a = 1, b = 2, c = 3)
* > x
* $a
* [1] 1
* $b
* [1] 2
* $c
* [1] 3
* a
* An interesting thing to note is that x$a gets you the actual element, and not a list of length 1 with 1 as the element. x[1] however, still gets you a list with 1 as the only element

### Setting Names of Matrices

* We can set the names of matrices using the dimnames() function. We need to specify a list , not vector of names though
* > m <- matrix(1:4, ncol=2, nrow=2)
* > dimnames(m) <- list(c("a", "b"), c("c", "d"))
* > m
* c d
* a 1 3
* b 2 4

## Reading Tabular Data

### Primary Reading Functions

* read.table() and read.csv() are used for reading tabular data. They read text files that have a table form. They both return a data frame
* readLines() is used for reading lines of a text file
  + Can read any type of file really
* The source() function is important for reading R code files and getting functions out of them
* The dget() function is also used for reading R code files but it’s meant for reading R objects that have been deparsed
* load() and unserialize() are used for reading binary objects into R

### Writing Data

* write.table()
* writeLines()
* dump()
* dput()
* save()
* serialize()
* All these functions pair up with their reading analogs described in the previous subsection

### read.table() paramters

* The most common function for reading data into R
* Here are the important arguments:
  + file, the name of a file or a connection
  + header, logical value indicating whether or not the file has a header
  + sep, a string indicating how columns are separated
  + colClasses, a character vector indicating the class of each column in the dataset
    - optional
  + nrows, the number of rows in the dataset
    - optional
  + comment.char, a character string indicating the comment character in the file
    - optional
  + skip, the number of lines to skip from the beginning
    - optional
  + stringsAsFactors, should character variables be coded as factors?
    - By default this is true

### Using read.table()

* For moderately sized datasets, we usually only need to specify the file name as R will take care of the rest for us.
* read.csv() is identical to read.table() except that the default separator is a comma, and it always specifies header to be true

## Reading Large Tables

* Read the help page for read.table()
* Figure out how much memory is required to store the dataset
* Set comment.char = “” if there are no commented lines in the file

### colClasses Argument

* This argument is very important since using it can make read.table() run much faster than not specifying anything
* If you don’t specify it, R by default goes through every column of the dataset and tries to figure out what data type it is. This process can take a long time for large datasets
* Obviously if you don’t know that data types ahead of time, what you can do to determine this is read in only 1000 rows let’s say by setting nrows = 1000. Then we can loop over each column using sapply() and calling the class() function on each column. Then we can store this vector and use it to specify the colClasses argument when we read in the entire dataset
* > initial <- read.table("datatable.txt", nrows=100)
* > classes <- sapply(initial, class)
* > all <- read.table("datatable.txt", colClasses=classes)

### nRows Argument

* Doesn’t necessarily make R run any faster, but it does help with memory usage.
* If you can tell R the number of rows, it can calculate the memory that is required and not have to figure it out on the go

### Know Thy System

* How much memory is available
* What other applications are in use
* Are other people logged on to the system
* What is the operating system
* Is the OS 32 bit or 64 bit?

### Calculating Memory Requirements

Example

* W have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly how much memory is required to store this data frame?

1,500,000 x 120 x 8bytes/numeric

= 1440000000 bytes

= 1.34GB

* Since there exists quite a bit of overhead for reading things into memory, the rule of thumb is that you need almost twice as much memory in order to read the dataset into R. I.e. 2.68 GB in this case

## Textual Data Formats

### Main Functions

* The two main functions for writing out data are dump() and dput()
* They result in text formats, but they are different than tables since they contain additional metadata.
  + For example, if you dump() or dput() a data frame, it will include in the output the class of each column so that you don’t have to specify it again when you read it in
* The advantage of using this type of text format is saving time when reading the data in again.
* The functions source() and dget() are the reading equivalents
* Textual formats are editable so if something gets corrupted, you can look at the file and see if you can recover it. This makes textual formats longer lived
* Also good for SVN so that you can track changes between documents
* The downside is that they aren’t space efficients.

### Dputting() R Objects

* One way to pass data around is by deparsing an R object with dput() and then reading it back in using dget()

#### Example

> y <- data.frame(a = 1, b = "a")

> dput(y)

structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), .Names = c("a",

"b"), row.names = c(NA, -1L), class = "data.frame")

> dput(y, "y.R")

> new.y <- dget("y.R")

> new.y

a b

1 1 a

* Notice the metadata like row.names and the class of the object as well

### Dumping R Objects

* The main difference between dput() and dump() is that dump() makes it possible to write **multiple** R objects whereas dput() allows only 1

#### Example

> x <- "foo"

> y <- data.frame(a = 1, b = "a")

> dump(c("x", "y"), file="data.R")

> rm(x, y)

> source("data.R")

> y

a b

1 1 a

> x

[1] "foo"

* Notice the quotes in the vector that we specify to dump()

## Interfaces to the Outside World

### Idea

* We want to interface with things like websites and files, even compressed files
* file(), opens a connection to a file
* gzfile(), opens a connection to a file compressed with gzip
* bzfile(), opens a connection to a file compressed with bzip2
* url(), opens a connection to a webpage
  + You can read data from that webpage using that url connection

### File Connections

|  |
| --- |
| > str(file)  function (description = "", open = "", blocking = TRUE,  encoding = getOption("encoding"), raw = FALSE) |
|  |
|  |

* description is the name of the file
* open is acode indicating
  + “r” read only
  + “w” writing and initializaing a new file
  + “a” appending
  + “rb”, “wb”, “ab”, reading, writing, appending in binary mode in Windows

### Connections

* In general you don’t have to deal with the connection interface

#### Example

> con <- file("foo.txt", "r")

> data <- read.csv(con)

> close(con)

* Is the same as the following

> data <- read.csv("foo.txt")

* So if you just want to read in all of the contents of a file, you’re better off just using the latter mnethod

### Reading Lines of a Text File

#### Example

> con <- gzfile("words.gz")

> x <- readLines(con, 10)

* This example allows you to read in 10 llines from the file
* writeLines() takes a character vector and writes each element one line at a time to a text file

### Reading Lines of a Webpage

#### Example

> con <- url("http://www.jhsph.edu", "r")

> x <-readLines(con)

> head(x)

[1] "<!DOCTYPE html>"

[2] "<html lang=\"en\">"

[3] ""

[4] "<head>"

[5] "<meta charset=\"utf-8\" />"

[6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"

## Subsetting Basics

### Operators

* [ always return an object of the same class as the original (i.e. if you subset a vector, you’re gonna get back a vector, if you subset a list, you’re gonna get back a list), can be sed to select more than one element (there is one exception)
* [[ is used to extract elements of a list or a data frame; it an only be used to extract a single element. The class of the returned object will not necessarily be a list or a data frame
* $ Is used to extract elements of a list or a data frame by name, smeaticcs are similar to that of [[

### Vectors

#### Example with Numeric Index

> x <- c("a", "b", "c", "d","a")

> x[1]

[1] "a"

> x[2]

[1] "b"

> x[1:4]

[1] "a" "b" "c" "d"

* In these three examples, we have subset the vector x using a numeric index
* Another type of index we can use is a logical index

#### Example with Logical Index

> x[x > "a"]

[1] "b" "c" "d"

> u <- x > "a"

> u

[1] FALSE TRUE TRUE TRUE FALSE

> x[u]

[1] "b" "c" "d"

## Subsetting Lists

* Subsetting a list is a little bit different thatn subsetting a vector since we can use the [[ or the $ operator as well
* You can also use the [ operator

#### Example With [

> x <- list(foo=1:4, bar=0.6)

> x[1]

$foo

[1] 1 2 3 4

* So remember that the [ operator always returns an element that is the same class as the original. In this case since x is a list, we get back a list when doing x[1]

#### Example With [[

> x[[1]]

[1] 1 2 3 4

* What we get back is the sequence 1:4
* The difference between this example and the previous example is that in the first example we got back a list containing the seuquence 1:4 whereas here we get back just the sequence 1:4

#### Example With $

> x$bar

[1] 0.6

* We want the element with the name bar
* Notice how we get back just the numeric vector, and NOT a list containing the numeric vector

#### Examples With [ [[ and names

> x[["bar"]]

[1] 0.6

* Notice how this gives us just the value of 0.6 since we are using [[

> x["bar"]

$bar

[1] 0.6

* Notice how this gives us a list containing the value 0.6 since we are using the [ operator

#### Example Extracting Multiple Elements

* If we want to get multiple elements, we need to use the [ operator
* CANNOT use [[ or $ when you want to extract multiple elements of a list

> x <- list(foo = 1:4, bar = 0.6, baz = "hello")

> x[c(1,3)]

$foo

[1] 1 2 3 4

$baz

[1] "hello"

> class(x[c(1,3)])

[1] "list"

* This example simply gets us the first and third elements of the list
* But notice how what we get back is a list since we are using the [ operator which always gives us back a result of the same class as the object we’re subsetting

### [[ vs $

* The nice thing about [[ is that it can be used with computed indices; $ can only be used with literal names

#### Example

> x <- list(foo = 1:4, bar = 0.6, baz = "hello")

> name <- "foo"

> x[[name]]

[1] 1 2 3 4

> x$name

NULL

> x$foo

[1] 1 2 3 4

* Sometimes the name of the element is actually the result of some computation
* [[ allows us to use that result to subset our list; $ does not allow us to use a variable to susbset
* See how x$name didn’t work even! Even though name = “foo”

### Subsetting Nested Elements of a List

* The [[ operator can take an integer sequence
* Suppose we want to extract the number 14
  + Well that’s the third element of the first element. By first element in this case we main the list that is containing the inner list

#### Example

> x <- list(a = list(10,12,14), b=c(3.14,2.81))

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

[1] 14

> x[[1]][[3]]

* So we can think of the operator as recursing into the list

## Subsetting Matrices

* Matrices can be subset the usual way with (i,j) type indices

#### Example

> x <- matrix(1:6, 2, 3)

> x[1, 2]

[1] 3

> x[2, 1]

[1] 2

* So the first index is the row index, the second index is the column index

#### Example with Missing Index

> x[1,]

[1] 1 3 5

> x[,2]

[1] 3 4

* Notice how x[1, ] gives us the first ROW
* X[, 2] gives us the second COLUMN

### Further Details

* By default when a single element of a matrix is retireuved, it is returned as a vector of length 1 rather than a 1x1 matrix. This is the exception to the [ always returning the same type as the object it indexes.
* This behaviour can be turned off by setting drop = FALSE
* > x[1,2]
* [1] 3
* > x[1,2,drop=FALSE]
* [,1]
* [1,] 3
* Notice how the second statement gave us back a matrix
* The same thing happens when we subset a single column or a single row

## Subsetting Partial Matching

* Saves a lot of time when working on the command line
* The idea is that it works with [[ and $
* Suppose we have a list x that has an element named aardvark in it

#### Example

> x <- list(aarvard = 1:5)

> x$a

[1] 1 2 3 4 5

> x[["a"]]

NULL

> x[["a", exact=F]]

[1] 1 2 3 4 5

* The way that $ works by default is that it looks for a name in the list that matches the letter “a”. In this case we only have one element so we get the object associated with “aardvark”
* The [[ operator by default wants to do EXACT matching so x[[“a”]] returns null. We need to set exact=FALSE if we want it to work like the $ operator
* Another interesting thing to note is that if you have two list elements both starting with “a” and you try to do x$a, you will get back null. Same thing with happen if you do x[[“a”, exact=F]]

## Subsetting Removing Missing Values

### Basics

* A common task is removing NAs from data
* Most realistic data has a lot of missing values
* The way to do this for a vector, matrix, or data frame is to create a logical vector which tells us where the NAs are so that we can remove them by subsetting
* We get this logical vector via the is.na() function

#### Example

> x <- c(1,2, NA, 4, NA, 5)

> bad <- is.na(x)

> bad

[1] FALSE FALSE TRUE FALSE TRUE FALSE

> x[!bad]

[1] 1 2 4 5

* He first thing we do is use the is.na() function to get the logical vector that tells us which elements in our vector x are NA
* So bad is our logical vector that tells us which elements are missing from x
* But we don’t want bad, we want the inverse of it since that tells us which elements are NOT missing

#### Example For Multiple Vectors

* What if there are multiple things and you want to take the subset with no missing values?

x <- c(1,2, NA, 4, NA, 5)

> y <- c("a", "b", NA, "d", NA, "f")

> good <- complete.cases(x,y)

> good

[1] TRUE TRUE FALSE TRUE FALSE TRUE

> x[good]

[1] 1 2 4 5

> y[good]

[1] "a" "b" "d" "f"

* So what the complete.cases() function does is say of the 2 vectors, which positions contain values in both vectors.

### Removing Missing Values From Data Frames

#### Example

> airquality[1:6,]

Ozone Solar.R Wind Temp Month Day

1 41 190 7.4 67 5 1

2 36 118 8.0 72 5 2

3 12 149 12.6 74 5 3

4 18 313 11.5 62 5 4

5 NA NA 14.3 56 5 5

6 28 NA 14.9 66 5 6

> good <- complete.cases(airquality)

> airquality[good, ][1:6, ]

Ozone Solar.R Wind Temp Month Day

1 41 190 7.4 67 5 1

2 36 118 8.0 72 5 2

3 12 149 12.6 74 5 3

4 18 313 11.5 62 5 4

7 23 299 8.6 65 5 7

8 19 99 13.8 59 5 8

* So notice how in the second subset we skip over days 4 and 5 and have days 7 and 8 instead

## Vectorized Operations

* Many operations in R are vectorixzed making code more efficient, concise, and eaier to read
* In other languages, if you have 2 vectors, you probable have to run a loop over them in order to add them component-wise

#### Example with Vectors

> x <- 1:4

> y <- 6:9

> x + y

[1] 7 9 11 13

* Notice how this add the vecotrs component-wise. I.e. it results in (x[1] + y[1], x[2] + y[2], x[3] + y[3], x[4] + y[4])

> x > 2

[1] FALSE FALSE TRUE TRUE

* Notice how since the operation is vectorized, we end up comparing all numbers to 2 and get a logical vector that’s the same length as x in return

> x >= 2

[1] FALSE TRUE TRUE TRUE

> y == 8

[1] FALSE FALSE TRUE FALSE

> x \* y

[1] 6 14 24 36

> x / y

[1] 0.1666667 0.2857143 0.3750000 0.4444444

> x ^ y

[1] 1 128 6561 262144

#### Example with Matrices

> x <- matrix(1:4,2,2)

> y <- matrix(rep(10, 4),2,2)

> x \* y

[,1] [,2]

[1,] 10 30

[2,] 20 40

* Notice how this gives us component-wise matrix multiplication

> x / y

[,1] [,2]

[1,] 0.1 0.3

[2,] 0.2 0.4

>

> x %\*% y

[,1] [,2]

[1,] 40 40

[2,] 60 60

* We need to use the %\*% operator to get true matrix multiplication

### Further Notes

If you have a vector z <- c(1.1, 9, 3.14)

When we did z \* 2 + 100 in our earlier example, z was a vector of length 3, but

| technically 2 and 100 are each vectors of length 1.

Behind the scenes, R is 'recycling' the 2 to make a vector of 2s and the 100 to make a

| vector of 100s. In other words, when you ask R to compute z \* 2 + 100, what it really

| computes is this: z \* c(2, 2, 2) + c(100, 100, 100).

# Week 2

## Control Structures - If-else

* The syntasx is very similar to that of C# in that is uses curly braces
* Has else-if just like C#

#### Example

> if (x > 3) {

+ y <- 10

+ } else {

+ y <- 0

+ }

* Notice how the assignment of y is happening on the inside of the conditional statements

This is equivalent to doing

> y <- if (x > 3) {

+ 10

+ } else {

+ 0

+ }

So here you’re basically saying that y is equal to the entire if else construct

Sometimes it’s sueful to use this syntax since it lets you know that the entire if+else construct is about assigning a value to y

## Control Strucutres - For Loop

* for loops in R take an iterator variable and assign it succeccive values from a sequence or a vector.

#### Example

> for (i in 1:10){

+ print(i)

+ }

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

[1] 6

[1] 7

[1] 8

[1] 9

[1] 10

> i

[1] 10

So notice how the variable i still exists after the loop has finished and now contains the last value that it took on while in the loop. This is different from a lot of other languages where the iterator variable only exists within the loop itself.

#### Example with Multiple Ways of Iterating

> x <- c("a","b","c","d")

##### Standard Way

> for(i in 1:4) {

+ print(x[i])

+ }

[1] "a"

[1] "b"

[1] "c"

[1] "d"

##### Using seq\_along()

> for (i in seq\_along(x)) {

+ print(x[i])

+ }

[1] "a"

[1] "b"

[1] "c"

[1] "d"

* What the seq\_along() function does is take a vector as input and creates an integer sequence the length of that vector
* > seq\_along(x)
* [1] 1 2 3 4

##### Using a non-integer iterator

>

> for (letter in x) {

+ print(letter)

+ }

[1] "a"

[1] "b"

[1] "c"

[1] "d"

* There’s no reason why the index variable has to be an integer
* In this example, the index variable itself takes on the values from our vector x and we can print it directly

##### Single Line For Loop

>

> for (i in 1:4) print(x[i])

[1] "a"

[1] "b"

[1] "c"

[1] "d"

* You don’t need curly braces if the body of your for loop only spans a single line

### Nested For Loops

* For loops can be nested, but doing more than 3 levels of nesting makes it difficult to understand
* s

#### Example Iterating Over Matrix

> x <- matrix(1:6, 2, 3)

> for (i in seq\_len(nrow(x))) {

+ for (j in seq\_len(ncol(x))) {

+ print(x[i,j])

+ }

+ }

[1] 1

[1] 3

[1] 5

[1] 2

[1] 4

[1] 6

* The seq\_len() function takes an integer and creates an integer sequence out of that

## Control Structures – Repeat, Next, Break

### Repeat and Break

#### Basics

* repeat is a construct that initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a repeat loop is to call break

#### Example

> x0 <- 1

> tol <- 1e-8

> repeat {

+ x1 <- computeEstimate()

+

+ if (abs(x1 - x0) < tol) {

+ break

+ } else {

+ x0 <- x1

+ }

+ }

#### Problems

* The loop in the above example has no guarantee that it will end. This is dangerous
* It’s better to set a hard limit on the number of iterations (using a for loop) and then report whether convergence was achieved or not

### Next

* next is used to skip an iteration of a loop

#### Example

> for (i in 1:100) {

+ if (i <= 20) {

+ ## skip the first 20 iterations

+ next

+ }

+

+ ## Do something here

+ }

### Return

* return signals that a function should exit and return a given value

## First R Function

### Syntax Notes

* Functions are created using the “function” directive.
* We assign to a variable the result of this function directive
* No return types need to be specified, nor do the parameter types
* testFunction <- function(x) {print(x)} is a basic function that simply calls print on the parameter x that we pass to it

#### Default Arguments

* You can have a default value for an argument like so
* testFunction <- function(x =10) {print(x)}
* If you call testFunction() with no argument, it will just print 10

### Add 2 Values

* Here is a function that simply adds 2 values

add2 <- function(x, y) {

x + y

}

* One thing to note is that we’re not using a return statement here. This is because R returns the last expression in a function

> add2(4,5)

[1] 9

### Return Subset of Values Great Than n

* Here we create a function that takes in a vector and returns a subset of that vector, having only the values that are greater than 10

above <- function(x, n) {

use <- x > n

x[use]

}

> above(c(15, 20, 21), 20)

[1] 21

### Mean of Columns from Matrix or Data Frame

* This function is slightly more complicated because we have to loop through the argument calculate the mean of each column

columnmean <- function(y, removeNA = TRUE) {

nc <- ncol(y)

means <- numeric(nc)

for (i in 1:nc) {

means[i] <- mean(y[, i], na.rm = removeNA)

}

means

}

> columnmean(airquality)

[1] 42.129310 185.931507 9.957516 77.882353 6.993464 15.803922

* So we have a default argument in this function that specfies that the mean() function we call on each column of our variable should ignore NA values by default. If we don’t do this, then if there is a single NA value in the column, the mean will be NA as well

> columnmean(airquality, FALSE)

[1] NA NA 9.957516 77.882353 6.993464 15.803922

* Notice how when we call the function this time, with our removeNA value equal to false, the first two columns have means of NA because the data within them has NA values

### Final Notes

* When we’re done writing functions in R studio, what we want to do is then save them in a file for later use

## Functions Part 1

### Basics

* There is some overlap here with the previous lecture
* Functions are created using the function() directice and are stored as R objects just like anything else, they are R objects of class “function”
* Functions in R are “first class objects”, which means that they can be treated much like any other R object. Importantly,
  + Functions can be passed as arguments to other functions
  + Functions can be nested, so that you can define a function inside of another function. The return value of a function is the last expression in the function body to be evaluated

### Function Arguments

* Functions have named arugments which potentially have default values
* The formal arguments are the arguments included in the function definintion
* The formals() function returns a list of all the formal arugments of a function
* Not every function call in R makes use of all the formal arguments
* Function arguments can be missing or might have default values

### Argument Matching

* R function arguments can be matched positionally or by name. So the following calls to sd() are all equivalent

> mydata <- rnorm(100)

> sd(mydata)

[1] 0.9135463

> sd(x = mydata)

[1] 0.9135463

> sd(x = mydata, na.rm= FALSE)

[1] 0.9135463

> sd(na.rm = FALSE, x = mydata)

[1] 0.9135463

> sd(na.rm = FALSE, mydata)

[1] 0.9135463

* Even though it’s possible, it’s not reccommened to mess around with the order of the arguments
* In the last example where we name one arugment and don’t name the other, you can think of the named argument as being removed from the list and any other arguments are matched against the function arguments in the way that they come
  + You can mix positional matching with matching by name. When an argument is matched by name, it is “taken out” of the argument list and the remaining unnamed argumetns are matced in the order that they are listed in the function definition.

#### Example

* The following 2 calls to lm() are equivalent

> args(lm)

function (formula, data, subset, weights, na.action, method = "qr",

model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,

contrasts = NULL, offset, ...)

NULL

> lm(data = mydata, y ~ x, model = FALSE, 1:100)

> lm(y ~ x, mydata, 1:100, model = FALSE)

### Partial Argument Matching

* Fucntion arguments can also be partially matched, which is useful for interactive work. The order of operations when given an argument is
  1. Check for exact match for a named argument
  2. Check for a partial match
  3. Check for a positional match
* This means that is a function has an argument with a very long name, you can type in just part of that name, so long as there is a unique match there

## Functions Part 2

### Defining a Function

* In addition to not specifying a default value, you can also wet an argument value to NULL

f <- function(a, b = 1, c = 2, d = NULL) {

}

### Lazy Evaluation

* One of the key features of the R language is something called lazy evaluation
* Arguments to functions are evaluated lazily, so they are evaluated only as needed

#### Example

> f <- function(a, b) {

a^2

}

> f(2)

[1] 4

* This function never actually uses the arugment b, so calling f(2) will not product an error because 2 gets positionally matched to a.

#### Example

f <- function(a, b) {

print(a)

print(b)

}

> f(45)

[1] 45

Error in print(b) : argument "b" is missing, with no default

* So notice how that the “45” got printed first, before the error was triggered. This is because b did not have to be evaualted until after print(a). Once the function

## Scoping Rules – Symbol Binding

### A Diversion on Binding Values to Symbol

* How does R know which value to assign to which symbol?
* If we type the following:

> lm <- function(x) { x \* x }

> lm

function(x) { x \* x }

* How does R know which value to assign to the symbol lm? Why doesn’t it give the value of lm that is in the stats package?
* When R tires to bind a value t a symbol, it searches through a series of environments to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly
  1. Search the global environment for a symbol name matching the requested one
  2. Search the namespaces of each of the packages on the search list
* The search list can be found by using the search() function like so

> search()

[1] ".GlobalEnv" "tools:rstudio" "package:stats" "package:graphics" "package:grDevices"

[6] "package:utils" "package:datasets" "package:methods" "Autoloads" "package:base"

* In this case, since we defined lm in the “.GlobalEnv” R finds that object first. If there is no match in the global environment, R will continue searching the namespaces of the remaining packages on the search list

### Biding Values to Symbol

* The global environment r the user’s workspace is always the first element in the search list and the base package is always the last
* The order of the packages in the search list matters
* User’s can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available
* When a user loads a package with the library() function, the namespace of that package gets put in position 2 of the search list
* Note that R has separate namespacs for functions and non0functions so it’s possible to have an objected named c and a function named c. Mind you in the global environment there can only be one symbol named c

### Scoping Rules

* The scoping rules for R are the main feature that make is different from the original S language
* The scoping rules determine how a value is associated with a free variable in a function
  + If you’re in a function, there are 2 types of variables: function arguments, other symbols found in the function that are not function arguments
* R uses lexical scoping or static scoping. A common alternative is dynamic scoping
* Related to the scoping rules is how R uses the search list to bind a value to a symbol
* Lexical scooping turns out to be particularly useful for simplifying statistical computations

### Lexical Scoping Conceptually

Consider the following function

f <- function(x, y) {

x^2 + y / z

}

* This functionhas 2 formal arguments x and y. In the boy of the function there is another symbol z. In this case z is called a free variable
* The scoping rules of a language determine how values are assigned to free variales.
* Free variables are not formal arguments and are not local variables assigned inside the function body

### Lexical Scoping in Practice

Lexical scoping in R means that:

* The values of free variables are searched for in the environment in which the function was defined

#### What is an environment?

* An environment is a collection of (symbol, value) pairs, i.e. x is a symbol and 3.14 might be its value
* Every environment has a parent environment; it is possible for an environment to have multiple children
* The only environment without a parent is the empty environment
* A function + and environment = a closure or function closure