Introduction to R

Presented by:

BROWN
School of Public Health

BROWN
School of Public Health

Intro to R Programming for Biostatistics

Day 1 - Getting Data in R

Adam J Sullivan

Lists

Lists

- Within R a list is a structure that can combine objects of different types.
- We will learn how to create and work with lists in this section.

Creating Lists

- A list is actually a vector but it does differ in comparison to the other types of vectors which we have been using in this class.
    - Other vectors are *atomic vectors*
    - A list is a type of vector called a *recursive vector*.

An Example Database

We first consider a patient database where we want to store their

- Name
- Amount of bill due
- A Boolean indicator of whether or not they have insurance.

Types of Information

We then have 3 types of information here:

- character
- numerical
- logical.

Single Patient

To create a list of one patient we say

```
a <- list(name="Angela", owed="75", insurance=TRUE)
a
```

```
## $name
## [1] "Angela"
##
## $owed
## [1] "75"
##
## $insurance
## [1] TRUE
```

Indexing

- Notice that unlike a typical vector this prints out in multiple parts.
- This also allows us to help with indexing as we will see below.
- There is another easy way to create this same list

Creating the Same List

```
a.alt <- vector(mode="list")
a.alt[["name"]] <- "Angela"
a.alt[["owed"]] <- 75
a.alt[["insurance"]] <- TRUE

a.alt
```

```
## $name
## [1] "Angela"
##
## $owed
## [1] 75
##
## $insurance
## [1] TRUE
```

Lists of Lists

- We could then create a list like this for all of our patients.
- Our database would then be a list of all of these individual lists.

List Operations

- With vectors, arrays and matrices, there was really only one way to index them.
- However with lists there are multiple ways:

List Indexing

```
a[["name"]]
```

```
## [1] "Angela"
```

```
a[[1]]
```

```
## [1] "Angela"
```

```
a$name
```

```
## [1] "Angela"
```

Double vs Single Brackets

- All of the previous are ways to index data in a list.
- Notice that in two of the above we used double brackets.
- Next we see the difference between double and single brackets.

Double vs Single Brackets

```
a[1]
```

```
## $name
## [1] "Angela"
```

```
class(a[1])
```

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

Double vs Single Brackets

```
a[[1]]
```

```
## [1] "Angela"
```

```
class(a[[1]])
```

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

Double vs Single Brackets

- With the single bracket we are returned another list.
- With the double bracket we are returned an element in the original class of what kind of data we entered.
- Depending on your goals you may want to use single or double brackets.

Adding and Subtracting Elements

With a list we can always add more information to it.

```
a$age <- 27
a
```

```
## $name
## [1] "Angela"
##
## $owed
## [1] "75"
##
## $insurance
## [1] TRUE
##
## $age
## [1] 27
```

Adding and Subtracting Elements

In order to delete an element from a list we set it to NULL.

```
a$owed <- NULL
a
```

```
## $name
## [1] "Angela"
##
## $insurance
## [1] TRUE
##
## $age
## [1] 27
```

List Components and Values

In order to know what kind of information is included in a list we can look at the ***names()*** function

```
names(a)
```

```
## [1] "name"      "insurance" "age"
```

Unlisting

To find the values of things we could go ahead and unlist them

```
a.un <- unlist(a)

a.un
```

```
##      name insurance       age
##  "Angela"    "TRUE"      "27"
```

```
class(a.un)
```

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

Unlisting

- If There is Character data in the original list that unlisted everything will be in character format.
- If your list contained all numerical elements than the class would be numerical.

Applying Functions to Lists

- Just like arrays and matrices we can use an ***apply()*** function.
- Specifically we have ***lapply()*** and ***sapply()*** functions for lists.
- With the original ***apply()*** function we could specify whether the function was applied to either the rows or the columns.
- With the case of lists both functions are applied to elements of the list.

Applying Functions to Lists

```
#Number list
n <- list(1:5, 6:37)
n
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
##  [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
## [24] 29 30 31 32 33 34 35 36 37
```

Applying Functions to Lists

- With this list we see that we have two separate vectors of numbers included.
- Then let us see the results of either using ***lapply()*** and ***sapply()***

Applying Functions to Lists

```
lapply(n, median)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 21.5
```

```
sapply(n, median)
```

```
## [1]  3.0 21.5
```

Apply Functions an Lists

- The ***lapply()*** function returns a list with the median of each of the original lists.
- While the ***sapply()*** function returns a vector of the medians.

Recursive Lists

- Before it was mentioned that a list is a recursive vector.
- This is because we can actually have lists within lists.

Recursive Lists

For example let us go back to our patient data.

```
s <- list(name="Chandra", insurance="TRUE", age=36)

patients <- list(a,s)
patients
```

Recursive Lists

```
## [[1]]
## [[1]]$name
## [1] "Angela"
##
## [[1]]$insurance
## [1] TRUE
##
## [[1]]$age
## [1] 27
##
##
## [[2]]
## [[2]]$name
## [1] "Chandra"
##
## [[2]]$insurance
## [1] "TRUE"
##
## [[2]]$age
## [1] 36
```

Final Notes on Lists

- It is important to remember how we can call these features of lists.
- Many of you will want to use R for model building and regressions.
- You almost never want to use the generated output from R.
- For example R does not automatically return the confidence intervals with a regression.

Final Notes on Lists

- The output from most regression functions in R is actually a list.
- What this means is I can extract the elements from the list that I want in order to build tables that display the exact information that I want it to.
- This is why we take the time to discuss how to search what is in a list and how to access it.

Example with Output of a List

```
x <- rnorm(500,10, 3)
y <- 3*x + rnorm(500, 0, 2)
```

Example with Output of a List

```
fit <- lm(y~x)
fit
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)            x
##     -0.1676       3.0231
```

Example with Output of a List

- So R just gave me the coefficients back but no other information.
- This means my knowledge of accessing lists is key.

```
names(fit)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

Example with Output of a List

- I can see that R actually has a lot more information that they did not display for me.
- Next I consider a function where it summarizes the information from this model

Example with Output of a List

```
summary <- summary(fit)
summary
```

Example with Output of a List

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.2378 -1.3566 -0.1404  1.1708  5.2600
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.16761    0.31605   -0.53    0.596
## x            3.02309    0.03017  100.20   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.982 on 498 degrees of freedom
## Multiple R-squared:  0.9527, Adjusted R-squared:  0.9526
## F-statistic: 1.004e+04 on 1 and 498 DF,  p-value: < 2.2e-16
```

Example with Output of a List

```r
names(summary)
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

Conclusion of Lists

- R has so much information about regression that is never even displayed unless I dig deeper.
- Understanding lists and accessing information means you can output custom tables that look much more professional than what R gives you.

DataFrames in R

Dataframe

- With statistics we are most likely to use the data structure called a data frame.
- This is similar to a matrix in appearance however we can have multiple types of data in it like a list.
- Each column must contain the same type of data or R will most likely default to character for that column.
- It is very important that you become proficient in working with data frames in order to fully understand data analysis.

Creating Data Frames

We usually create a data frame with vectors.

```
names <- c("Angela", "Shondra")
ages <- c(27,36)
insurance <- c(TRUE, T)
patients <- data.frame(names, ages, insurance)
patients
```

```
##     names ages insurance
## 1  Angela   27      TRUE
## 2 Shondra   36      TRUE
```

Adding Rows or Columns

- We may wish to add rows or columns to our data.
- We can do this with:
    - ***rbind()***
    - ***cbind()***

Adding Rows or Columns

For example we can go back to our patient data and say we wish to add another patient we could just do the following

```
l <- c(names="Liu Jie", age=45, insurance=TRUE)
rbind(patients, l)
```

```
## Warning in `[<-.factor`(`*tmp*`, ri, value = "Liu Jie"): invalid factor
## level, NA generated
```

```
##     names ages insurance
## 1  Angela   27      TRUE
## 2 Shondra   36      TRUE
## 3    <NA>   45      TRUE
```

- This warning serves as a reminder to always know what your data type is.
- R has read our data in as a factor when we want it as a character.

Adding Rows or Columns

```
patients$names <- as.character(patients$names)
patients <- rbind(patients, l)
patients
```

```
##     names ages insurance
## 1  Angela   27      TRUE
## 2 Shondra   36      TRUE
## 3 Liu Jie   45      TRUE
```

Adding Rows or Columns

Finally if we decided to then place another column of data in we could

```
# Next appointments
next.appt <- c("09/23/2016", "04/14/2016", "02/25/2016")

#Lets R know these are dates
next.appt <- as.Date(next.appt, "%m/%d/%Y")
next.appt
```

```
## [1] "2016-09-23" "2016-04-14" "2016-02-25"
```

```r
patients <- cbind(patients, next.appt)
patients
```

```
##      names ages insurance  next.appt
## 1  Angela   27      TRUE 2016-09-23
## 2 Shondra   36      TRUE 2016-04-14
## 3 Liu Jie   45      TRUE 2016-02-25
```

Accessing Data Frames

In order to best consider accessing of data frames we will use some built in data from R.

```
library(datasets)
titanic <- data.frame(Titanic)
```

Variables Included in Titanic

```
colnames(titanic)
```

```
## [1] "Class"    "Sex"      "Age"      "Survived" "Freq"
```

Preview Into Data

```
titanic[1:2,]
```

```
##   Class  Sex   Age Survived Freq
## 1   1st Male Child       No    0
## 2   2nd Male Child       No    0
```

```
head(titanic)
```

```
##   Class    Sex   Age Survived Freq
## 1   1st   Male Child       No    0
## 2   2nd   Male Child       No    0
## 3   3rd   Male Child       No   35
## 4  Crew   Male Child       No    0
## 5   1st Female Child       No    0
## 6   2nd Female Child       No    0
```

Indexing

- Indexing is the same as that for matrices.
- ***head()*** function allows us to access the first rows of the data frame.
- We can also access data by using both column and row information

Indexing

```
# accessing age information
titanic[,3]
```

```
##  [1] Child Child Child Child Child Child Child Child Adult Adult Adult
## [12] Adult Adult Adult Adult Adult Child Child Child Child Child Child
## [23] Child Child Adult Adult Adult Adult Adult Adult Adult Adult
## Levels: Child Adult
```

```
#accessing age information using column name
titanic[, "Age"]
```

```
##  [1] Child Child Child Child Child Child Child Child Adult Adult Adult
## [12] Adult Adult Adult Adult Adult Child Child Child Child Child Child
## [23] Child Child Adult Adult Adult Adult Adult Adult Adult Adult
## Levels: Child Adult
```

Indexing and Naming

- This means we can access data with a column or row number
- More importantly we can use the name.
- For large data frames accessing by a name is key.

Further Indexing

We could ask for information by using the factors that we have as well

```
first.class.freq <- titanic[titanic$Class=="1st", "Freq"]
first.class.freq
```

```
## [1]   0   0 118   4   5   1  57 140
```

```
male.freq <- titanic[titanic$Sex=="Male", "Freq"]
male.freq
```

```
##  [1]   0   0  35   0 118 154 387 670   5  11  13   0  57  14  75 192
```

Our New Variables

```
sum(first.class.freq)
```

```
## [1] 325
```

```
sum(male.freq)
```

```
## [1] 1731
```

Adding New Variables

- Suppose we not only want to know the frequency of survival but the proportion
- We can ask R to calculate this and add it to our data.

```
titanic$surv_p <- titanic$Freq/sum(titanic$Freq)
head(titanic,4)
```

```
##   Class  Sex   Age Survived Freq     surv_p
## 1   1st Male Child       No    0 0.00000000
## 2   2nd Male Child       No    0 0.00000000
## 3   3rd Male Child       No   35 0.01590186
## 4  Crew Male Child       No    0 0.00000000
```

Replacing Values

- Perhaps we were not pleased the decimal places and want to have this as a percentage.
- We can overwrite the values and change this.

```
titanic$surv_p <- titanic$surv_p*100
head(titanic,4)
```

```
##   Class  Sex   Age Survived Freq   surv_p
## 1   1st Male Child       No    0 0.000000
## 2   2nd Male Child       No    0 0.000000
## 3   3rd Male Child       No   35 1.590186
## 4  Crew Male Child       No    0 0.000000
```

Tibbles in R

Tibbles

Previously we have worked with data in the form of

- Vectors
- Lists
- Arrays
- Dataframes

Tibbles

- *"Tibbles"* are a new modern data frame.
- It keeps many important features of the original data frame.
- It removes many of the outdated features.

Compared to Data Frames

- A *tibble* never changes the input type.
  - No more worry of characters being automatically turned into strings.
- A tibble can have columns that are lists.
- A tibble can have non-standard variable names.
  - can start with a number or contain spaces.
  - To use this refer to these in a backtick.
- It only recycles vectors of length 1.
- It never creates row names.

Column-Lists

```
library(tidyverse)
## Warning: package 'tidyverse' was built under R version 3.3.2
## Warning: package 'ggplot2' was built under R version 3.3.2
## Warning: package 'tidyr' was built under R version 3.3.2
try <- tibble(x = 1:3, y = list(1:5, 1:10, 1:20))
try
## # A tibble: 3 × 2
##       x          y
##   <int>     <list>
## 1     1  <int [5]>
## 2     2 <int [10]>
## 3     3 <int [20]>


#try <- as_data_frame(c(x = 1:3, y = list(1:5, 1:10, 1:20)))
#try
# Leads to error
```

Non-Standard Names

```
names(data.frame(`crazy name` = 1))
## [1] "crazy.name"
names(tibble(`crazy name` = 1))
## [1] "crazy name"
```

Coercing into Tibbles

- A tibble can be made by coercing `as_tibble()`.
- This works similar to `as.data.frame()`.
- It works efficiently.

Coercing into Tibbles

```
l <- replicate(26, sample(100), simplify = FALSE)
names(l) <- letters

microbenchmark::microbenchmark(
  as_tibble(l),
  as.data.frame(l)
)
## Unit: microseconds
##              expr      min       lq     mean    median        uq      max
##      as_tibble(l)  299.879  337.363  385.665  368.3775  411.6635  775.132
##  as.data.frame(l) 1357.485 1504.300 1747.447 1578.1535 1826.2675 3112.575
##  neval cld
##    100  a
##    100   b
```

Tibbles vs Data Frames

There are a couple key differences between tibbles and data frames.

- Printing.
- Subsetting.

Printing

- Tibbles only print the first 10 rows and all the columns that fit on a screen. - Each column displays its data type.
- You will not accidentally print too much.

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
```

Printing

```
## # A tibble: 1,000 × 5
##                      a          b     c          d     e
##                 <dttm>     <date> <int>      <dbl> <chr>
## 1  2017-02-18 05:28:37 2017-03-08     1 0.02150370     f
## 2  2017-02-17 22:08:24 2017-03-08     2 0.08031493     k
## 3  2017-02-18 02:03:13 2017-03-07     3 0.11670172     u
## 4  2017-02-18 15:16:10 2017-03-08     4 0.24552337     h
## 5  2017-02-18 00:41:20 2017-03-04     5 0.11232662     b
## 6  2017-02-18 06:26:41 2017-03-08     6 0.52834632     m
## 7  2017-02-18 10:08:57 2017-03-15     7 0.78928491     v
## 8  2017-02-18 13:28:41 2017-03-15     8 0.80388276     h
## 9  2017-02-18 11:35:47 2017-03-18     9 0.45767339     d
## 10 2017-02-18 05:40:18 2017-02-24    10 0.18177950     t
## # ... with 990 more rows
```

Subsetting

- We can index a tibble in the manners we are used to
    - `df$x`
    - `df[["x"]]`
    - `df[[1]]`
- We can also use a `pipe` which we will learn about later.
    - `df %>% .$x`
    - `df %>% .[["x"]]`

Subsetting

```
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)

df$x
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
df[["x"]]
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
df[[1]]
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
```

Subsetting

```
df %>% .$x
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
df %>% .[["x"]]
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
df %>% .[[1]]
## [1] 0.6227033 0.7363213 0.8551199 0.9173554 0.5542486
```