Tutorial 3 - Manipulating data, lists, functions; if-else statements

Content

- More on data frames
- Lists
- Writing functions in R
- If-else statements

More on data frames

```
library(MASS)
head(Cars93, 3)
```

```
##   Manufacturer   Model    Type Min.Price Price Max.Price MPG.city
## 1        Acura Integra   Small      12.9  15.9      18.8       25
## 2        Acura  Legend Midsize      29.2  33.9      38.7       18
## 3         Audi      90 Compact      25.9  29.1      32.3       20
##   MPG.highway            AirBags DriveTrain Cylinders EngineSize
## 1          31               None      Front         4        1.8
## 2          25 Driver & Passenger      Front         6        3.2
## 3          26        Driver only      Front         6        2.8
##   Horsepower  RPM Rev.per.mile Man.trans.avail Fuel.tank.capacity
## 1        140 6300         2890             Yes               13.2
## 2        200 5500         2335             Yes               18.0
## 3        172 5500         2280             Yes               16.9
##   Passengers Length Wheelbase Width Turn.circle Rear.seat.room
## 1          5    177       102    68          37           26.5
## 2          5    195       115    71          38           30.0
## 3          5    180       102    67          37           28.0
##   Luggage.room Weight  Origin          Make
## 1           11   2705 non-USA Acura Integra
## 2           15   3560 non-USA  Acura Legend
## 3           14   3375 non-USA       Audi 90
```

Adding a column: `transform()` function

- `transform()` returns a new data frame with columns modified or added as specified by the function call

```
Cars93.metric <- transform(Cars93,
                           KMPL.city = 0.425 * MPG.city,
                           KMPL.highway = 0.425 * MPG.highway)
tail(names(Cars93.metric))
```

```
## [1] "Luggage.room" "Weight"       "Origin"       "Make"
## [5] "KMPL.city"    "KMPL.highway"
```

- Our data frame has two new columns, giving the fuel consumption in km/l

Another approach

```
# Add a new column called KMPL.city.2
Cars93.metric$KMPL.city.2 <- 0.425 * Cars93$MPG.city
tail(names(Cars93.metric))
```

```
## [1] "Weight"       "Origin"       "Make"         "KMPL.city"
## [5] "KMPL.highway" "KMPL.city.2"
```

- Let's check that both approaches did the same thing

```
identical(Cars93.metric$KMPL.city, Cars93.metric$KMPL.city.2)
```

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

Changing levels of a factor

```
manufacturer <- Cars93$Manufacturer
head(manufacturer, 10)
```

```
##  [1] Acura    Acura    Audi     Audi     BMW      Buick    Buick
##  [8] Buick    Buick    Cadillac
## 32 Levels: Acura Audi BMW Buick Cadillac Chevrolet Chrylser ... Volvo
```

We'll use the `mapvalues(x, from, to)` function from the `plyr` library.

```
library(plyr)

# Map Chevrolet, Pontiac and Buick to GM
manufacturer.combined <- mapvalues(manufacturer,
                                   from = c("Chevrolet", "Pontiac", "Buick"),
                                   to = rep("GM", 3))

head(manufacturer.combined, 10)
```

```
##  [1] Acura    Acura    Audi     Audi     BMW      GM       GM
##  [8] GM       GM       Cadillac
## 30 Levels: Acura Audi BMW GM Cadillac Chrylser Chrysler Dodge ... Volvo
```

Another example

- A lot of data comes with integer encodings of levels

- You may want to convert the integers to more meaningful values for the purpose of your analysis
- Let's pretend that in the class survey 'Program' was coded as an integer with 1 = MISM, 2 = Other, 3 = PPM

```
survey <- read.table("http://www.andrew.cmu.edu/user/achoulde/94842/data/survey_data.csv", heade
r=TRUE, sep=",")
survey <- transform(survey, Program=as.numeric(Program))
head(survey)
```

```
##   Program               PriorExp      Rexperience OperatingSystem TVhours
## 1       1    Extensive experience Basic competence         Windows       0
## 2       3         Some experience       Never used         Windows       3
## 3       3         Some experience Basic competence        Mac OS X      30
## 4       2         Some experience Basic competence        Mac OS X       6
## 5       1         Some experience Basic competence        Mac OS X      20
## 6       3 Never programmed before       Never used         Windows      15
##            Editor
## 1  Microsoft Word
## 2  Microsoft Word
## 3  Microsoft Word
## 4 Microsoft Excel
## 5           LaTeX
## 6  Microsoft Word
```

Example continued

- Here's how we would get back the program codings using the `transform()`, `as.factor()` and `mapvalues()` functions

```
survey <- transform(survey,
                    Program = as.factor(mapvalues(Program,
                                                  c(1, 2, 3),
                                                  c("MISM", "Other", "PPM")))
                    )
head(survey)
```

```
##   Program               PriorExp      Rexperience OperatingSystem TVhours
## 1    MISM    Extensive experience Basic competence         Windows       0
## 2     PPM         Some experience       Never used         Windows       3
## 3     PPM         Some experience Basic competence        Mac OS X      30
## 4   Other         Some experience Basic competence        Mac OS X       6
## 5    MISM         Some experience Basic competence        Mac OS X      20
## 6     PPM Never programmed before       Never used         Windows      15
##            Editor
## 1  Microsoft Word
## 2  Microsoft Word
## 3  Microsoft Word
## 4 Microsoft Excel
## 5           LaTeX
## 6  Microsoft Word
```

Some more data frame summaries: `table()` function

- Let's revisit the Cars93 dataset
- The `table()` function builds **contingency tables** showing counts at each combination of factor levels

```
table(Cars93$AirBags)
```

```
##
## Driver & Passenger        Driver only               None
##                 16                  43                 34
```

```
table(Cars93$Origin)
```

```
##
##     USA non-USA
##      48      45
```

```
table(Cars93$AirBags, Cars93$Origin)
```

```
##
##                     USA non-USA
##   Driver & Passenger   9       7
##   Driver only         23      20
##   None                16      18
```

- Looks like US and non-US cars had about the same distribution of AirBag types
- Later in the class we'll learn how to do a hypothesis tests on this kind of data

Alternative syntax

- When `table()` is supplied a data frame, it produces contingency tables for all combinations of factors

```
head(Cars93[c("AirBags", "Origin")], 3)
```

```
##              AirBags  Origin
## 1               None non-USA
## 2 Driver & Passenger non-USA
## 3        Driver only non-USA
```

```
table(Cars93[c("AirBags", "Origin")])
```

```
##                     Origin
## AirBags              USA non-USA
##   Driver & Passenger   9       7
##   Driver only         23      20
##   None                16      18
```

Basics of lists

> A list is a **data structure** that can be used to store **different kinds** of data

- Recall: a vector is a data structure for storing *similar kinds of data*
- To better understand the difference, consider the following example.

```
my.vector.1 <- c("Michael", 165, TRUE) # (name, weight, is.male)
my.vector.1
```

```
## [1] "Michael" "165"     "TRUE"
```

```
typeof(my.vector.1)  # All the elements are now character strings!
```

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

Lists vs. vectors

```
my.vector.2 <- c(FALSE, TRUE, 27) # (is.male, is.citizen, age)
typeof(my.vector.2)
```

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

- Vectors expect elements to be all of the same type (e.g., `Boolean` , `numeric` , `character`)
- When data of different types are put into a vector, the R converts everything to a common type

Lists

- To store data of different types in the same object, we use lists
- Simple way to build lists: use `list()` function

```
my.list <- list("Michael", 165, TRUE)
my.list
```

```
## [[1]]
## [1] "Michael"
##
## [[2]]
## [1] 165
##
## [[3]]
## [1] TRUE
```

```
sapply(my.list, typeof)
```

```
## [1] "character" "double"    "logical"
```

Named elements

```
patient.1 <- list(name="Michael", weight=165, is.male=TRUE)
patient.1
```

```
## $name
## [1] "Michael"
##
## $weight
## [1] 165
##
## $is.male
## [1] TRUE
```

Referencing elements of a list (similar to data frames)

```
patient.1$name # Get "name" element (returns a string)
```

```
## [1] "Michael"
```

```
patient.1[["name"]] # Get "name" element (returns a string)
```

```
## [1] "Michael"
```

```
patient.1["name"] # Get "name" slice (returns a sub-list)
```

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

```
c(typeof(patient.1$name), typeof(patient.1["name"]))
```

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

Functions

- We have used a lot of built-in functions: `mean()`, `subset()`, `plot()`, `read.table()` ...
- An important part of programming and data analysis is to write custom functions
- Functions help make code **modular**
- Functions make debugging easier
- Remember: this entire class is about applying *functions* to *data*

What is a function?

> A function is a machine that turns **input objects** (arguments) into an **output object** (return value) according to a definite rule.

- Let's look at a really simple function

```
addOne <- function(x) {
  x + 1
}
```

- `x` is the **argument** or **input**
- The function **output** is the input `x` incremented by 1

```
addOne(12)
```

```
## [1] 13
```

More interesting example

- Here's a function that returns a % given a numerator, denominator, and desired number of decimal values

```
calculatePercentage <- function(x, y, d) {
  decimal <- x / y  # Calculate decimal value
  round(100 * decimal, d)  # Convert to % and round to d digits
}

calculatePercentage(27, 80, 1)
```

```
## [1] 33.8
```

- If you're calculating several %'s for your report, you should use this kind of function instead of repeatedly copying and pasting code

Function returning a list

- Here's a function that takes a person's full name (FirstName LastName), weight in lb and height in inches and converts it into a list with the person's first name, person's last name, weight in kg, height in m, and BMI.

```
createPatientRecord <- function(full.name, weight, height) {
  name.list <- strsplit(full.name, split=" ")[[1]]
  first.name <- name.list[1]
  last.name <- name.list[2]
  weight.in.kg <- weight / 2.2
  height.in.m <- height * 0.0254
  bmi <- weight.in.kg / (height.in.m ^ 2)
  list(first.name=first.name, last.name=last.name, weight=weight.in.kg, height=height.in.m,
       bmi=bmi)
}
```

Trying out the function

```
createPatientRecord("Michael Smith", 185, 12 * 6 + 1)
```

```
## $first.name
## [1] "Michael"
##
## $last.name
## [1] "Smith"
##
## $weight
## [1] 84.09091
##
## $height
## [1] 1.8542
##
## $bmi
## [1] 24.45884
```

Another example: 3 number summary

- Calculate mean, median and standard deviation

```
threeNumberSummary <- function(x) {
  c(mean=mean(x), median=median(x), sd=sd(x))
}
x <- rnorm(100, mean=5, sd=2) # Vector of 100 normals with mean 5 and sd 2
threeNumberSummary(x)
```

```
##     mean   median       sd
## 4.903881 5.014473 1.746474
```

If-else statements

- Oftentimes we want our code to have different effects depending on the features of the input
- Example: Calculating a student's letter grade
- If grade >= 90, assign A
- Otherwise, if grade >= 80, assign B
- Otherwise, if grade >= 70, assign C
- In all other cases, assign F
- To code this up, we use if-else statements

If-else Example: Letter grades

```
calculateLetterGrade <- function(x) {
  if(x >= 90) {
    grade <- "A"
  } else if(x >= 80) {
    grade <- "B"
  } else if(x >= 70) {
    grade <- "C"
  } else {
    grade <- "F"
  }
  grade
}

course.grades <- c(92, 78, 87, 91, 62)
sapply(course.grades, FUN=calculateLetterGrade)
```

```
## [1] "A" "C" "B" "A" "F"
```

`return()`

- In the previous examples we specified the output simply by writing the output variable as the last line of the function
- More explicitly, we can use the `return()` function

```
addOne <- function(x) {
  return(x + 1)
}

addOne(12)
```

```
## [1] 13
```

- We will generally avoid the `return()` function, but you can use it if necessary or if it makes writing a particular function easier.

Next

- Complete Lab 3 (http://isle.heinz.cmu.edu/94-842/lab03/)