---
title: "Chapter 2: Data Classes, Functions"
author: "M Affouf"
date: "12/24/2017"
output:output: html_document

#part1

Data Classes:

* One dimensional classes ('vectors'):
    * Character: strings or individual characters, quoted
    * Numeric: any real number(s)
    * Integer: any integer(s)/whole numbers
    * Factor: categorical/qualitative variables
    * Logical: variables composed of TRUE or FALSE
    * Date/POSIXct: represents calendar dates and times

Character and numeric

We have already covered `character` and `numeric` classes.

```{r numChar}
# Change accordingly
class(c("Your First Name", "Your Last Name"))
class(c(1, 4, 7))
```

Integer

`Integer` is a special subset of `numeric` that contains only whole numbers

A sequence of numbers is an example of the integer class

```{r seq}
x = seq(from = 1, to = 5) # seq() is a function
x
class(x)
```

Integer

The colon `:` is a shortcut for making sequences of numbers

It makes consecutive integer sequence from `[num1]` to `[num2]` by 1


```{r seqShort}
1:5
```

Logical

`logical` is a class that only has two possible elements: `TRUE` and

`FALSE`

```{r logical1}
x = c(TRUE, FALSE, TRUE, TRUE, FALSE)
class(x)
```

`sum()` and `mean()` work on `logical` vectors - they return the total and proportion of `TRUE` elements, respectively.

Logical

Note that `logical` elements are NOT in quotes.
```{r logical2}
z = c("TRUE", "FALSE", "TRUE", "FALSE")
class(z)
```

Factor

`factor` are special `character` vectors where the elements have pre-defined groups or 'levels'. You can think of these as qualitative or categorical variables:

```{r factor1}
x = factor(c("boy", "girl", "girl", "boy", "girl"))
x
class(x)
```

Note that levels are, by default, alphabetical or alphanumerical order.

Factors

Factors are used to represent categorical data, and can also be used for ordinal data (ie categories have an intrinsic ordering)

Note that R reads in character strings as factors by default in functions like `read.table()`

'The function factor is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If argument ordered is TRUE, the factor levels are assumed to be ordered.'

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x))
```

Factors

Suppose we have a vector of case-control status

```{r factor2}
cc = factor(c("case","case","case",
```

        "control","control","control"))
cc
levels(cc) = c("control","case")
cc
```

## Factors

Note that the levels are alphabetically ordered by default. We can also specify the levels within the factor call

```{r factor_cc_again}
factor(c("case","case","case","control",
          "control","control"),
        levels =c("control","case") )
factor(c("case","case","case","control",
            "control","control"),
        levels =c("control","case"), ordered=TRUE)
```

Factors

Factors can be converted to `numeric` or `character` very easily

```{r factor3}
x = factor(c("case","case","case","control",
      "control","control"),
        levels =c("control","case") )
as.character(x)
as.numeric(x)
```

Factors

However, you need to be careful modifying the labels of existing factors, as its quite easy to alter the meaning of the underlying data.

```{r factorCheck}
xCopy = x
levels(xCopy) = c("case", "control") # wrong way
xCopy
as.character(xCopy) # labels switched
as.numeric(xCopy)
```

Creating categorical variables

the `rep()` ["repeat"] function is useful for creating new variables

```{r rep1}
bg = rep(c("boy","girl"),each=50)
head(bg)
bg2 = rep(c("boy","girl"),times=50)
head(bg2)
length(bg)==length(bg2)
```
```

Creating categorical variables

One frequently-used tool is creating categorical variables out of continuous variables, like generating quantiles of a specific continuously measured variable.

A general function for creating new variables based on existing variables is the `ifelse()` function, which "returns a value with the same shape as test which is filled with elements selected from either yes or no depending on whether the element of test is `TRUE` or `FALSE`."

```
ifelse(test, yes, no)

# test: an object which can be coerced
    to logical mode.
# yes: return values for true elements of test.
# no: return values for false elements of test.
```

Charm City Circulator data

Please download the Charm City Circulator data:

```{r}
circ = read.csv("Charm_City_Circulator_Ridership.csv",
            header=TRUE,as.is=TRUE)
```

Creating categorical variables

For example, we can create a new variable that records whether daily ridership on the Circulator was above 10,000.

```{r ifelse1}
hi_rider = ifelse(circ$daily > 10000, "high", "low")
hi_rider = factor(hi_rider, levels = c("low","high"))
head(hi_rider)
table(hi_rider)
```

Creating categorical variables

You can also nest `ifelse()` within itself to create 3 levels of a variable.

```{r ifelse2}
riderLevels = ifelse(circ$daily < 10000, "low",
                  ifelse(circ$daily > 20000,
                  "high", "med"))
riderLevels = factor(riderLevels,
              levels = c("low","med","high"))
head(riderLevels)
table(riderLevels)
```

Creating categorical variables

However, it's much easier to use `cut()` to create categorical variables from continuous variables.

'cut divides the range of x into intervals and codes the values in x according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.'

```
cut(x, breaks, labels = NULL, include.lowest = FALSE,
  right = TRUE, dig.lab = 3,
  ordered_result = FALSE, ...)
```

Creating categorical variables

`x`: a numeric vector which is to be converted to a factor by cutting.

`breaks`: either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which x is to be cut.

`labels`: labels for the levels of the resulting category. By default, labels are constructed using "(a,b]" interval notation. If `labels = FALSE`, simple integer codes are returned instead of a factor.

Cut

Now that we know more about factors, `cut()` will make more sense:

```{r cut1}
x = 1:100
cx = cut(x, breaks=c(0,10,25,50,100))
head(cx)
table(cx)
```

We can also leave off the labels

```{r cut2}
cx = cut(x, breaks=c(0,10,25,50,100), labels=FALSE)
head(cx)
table(cx)
```

Note that you have to specify the endpoints of the data, otherwise some of the categories will not be created

```{r cut3}
cx = cut(x, breaks=c(10,25,50), labels=FALSE)
head(cx)
table(cx)
table(cx,useNA="ifany")
```

Date

You can convert date-like strings in the `Date` class (http://www.statmethods.net/input/dates.html for more info)

```{r date}
head(sort(circ$date))
circ$newDate <- as.Date(circ$date, "%m/%d/%Y") # creating a date for sorting
head(circ$newDate)
range(circ$newDate)
```

Date

However, the `lubridate` package is much easier for generating explicit dates:

```{r}
library(lubridate) # great for dates!
suppressPackageStartupMessages(library(dplyr))
circ = mutate(circ, newDate2 = mdy(date))
head(circ$newDate2)
range(circ$newDate2)
```

POSIXct

The `POSIXct` class can encode time information

```{r}
theTime = Sys.time()
theTime
class(theTime)
theTime + 5000
```

Note it's like a more general date format.

Data Classes:

* Two dimensional classes:
    * `data.frame`: traditional 'Excel' spreadsheets
        * Each column can have a different class, from above
    * Matrix: two-dimensional data, composed of rows and columns. Unlike data frames, the entire matrix is composed of one R class, e.g. all numeric or all characters.

Matrices

```{r matrix}
n = 1:9
n
mat = matrix(n, nrow = 3)
mat
```

Matrix (and Data frame) Functions

These are in addition to the previous useful vector functions:

* `nrow()` displays the number of rows of a matrix or data frame
* `ncol()` displays the number of columns
* `dim()` displays a vector of length 2: # rows, # columns
* `colnames()` displays the column names (if any) and `rownames()` displays the row names (if any)

Data Selection

Matrices have two "slots" you can use to select data, which represent rows and columns, that are separated by a comma, so the syntax is `matrix[row,column]`. Note you cannot use `dplyr` functions on matrices.

```{r subset3}
mat[1, 1] # individual entry: row 1, column 1
mat[1, ] # first row
mat[, 1] # first columns
```

Data Selection

Note that the class of the returned object is no longer a matrix

```{r subset4}
class(mat[1, ])
class(mat[, 1])
```

Data Frames

To review, the `data.frame` is the other two dimensional variable class.

Again, data frames are like matrices, but each column is a vector that can have its own class. So some columns might be `character` and others might be `numeric`, while others maybe a `factor`.

Data Frames versus Matrices

You will likely use data.frame class for a lot of data cleaning and analysis. However, some operations that rely on matrix multiplication (like performing many linear regressions) are (much) faster with matrices. Also, as we will touch on later, some functions for iterating over data will return the matrix class, or will be placed in empty matrices that can then be converted to data.frames

Data Frames versus Matrices

There is also additional summarization functions for matrices (and not data.frames) in the `matrixStats` package, like `rowMins()`, `colMaxs()`, etc.

```{r}
library(matrixStats,quietly = TRUE)
```

```
avgs = select(circ, ends_with("Average"))
rowMins(as.matrix(avgs),na.rm=TRUE)[500:510]
```

Data Classes

Extensions of "normal" data classes:

* N-dimensional classes:
    * Arrays: any extension of matrices with more than 2 dimensions, e.g. 3x3x3 cube
    * Lists: more flexible container for R objects.

Arrays

These are just more flexible matrices - you should just be made aware of them as some functions return objects of this class, for example, cross tabulating over more than 2 variables and the `tapply` function.

Arrays

Selecting from arrays is similar to matrices, just with additional commas for the additional slots.

```{r}
ar = array(1:27, c(3,3,3))
ar[,,1]
ar[,1,]
```


Lists

* One other data type that is the most generic are `lists`.
* Can be created using list()
* Can hold vectors, strings, matrices, models, list of other list, lists upon lists!
* Can reference data using $ (if the elements are named), or using [], or [[]]

```{r makeList, comment="", prompt=TRUE}
mylist <- list(letters=c("A", "b", "c"),
        numbers=1:3, matrix(1:25, ncol=5))
```

List Structure
```{r Lists, comment="", prompt=TRUE}
head(mylist)
```

List referencing
```{r Listsref1, comment="", prompt=TRUE}
mylist[1] # returns a list
mylist["letters"] # returns a list
```

List referencing

```{r Listsrefvec, comment="", prompt=TRUE}
mylist[[1]] # returns the vector 'letters'
mylist$letters # returns vector
mylist[["letters"]] # returns the vector 'letters'
```

List referencing

You can also select multiple lists with the single brackets.

```{r Listsref2, comment="", prompt=TRUE}
mylist[1:2] # returns a list
```

List referencing

You can also select down several levels of a list at once

```{r Listsref3, comment="", prompt=TRUE}
mylist$letters[1]
mylist[[2]][1]
mylist[[3]][1:2,1:2]
```

Splitting Data Frames

The `split()` function is useful for splitting `data.frame`s

"`split` divides the data in the vector x into the groups defined by `f`. The replacement forms replace values corresponding to such a division. `unsplit` reverses the effect of split."

```{r split1, comment="", prompt=TRUE}
dayList = split(circ,circ$day)
```

Splitting Data Frames {.smaller}

Here is a good chance to introduce `lapply`, which performs a function within each list element:

```{r lapply1, comment="", prompt=TRUE}
# head(dayList)
lapply(dayList, head, n=2)
```

```{r lapply2, comment="", prompt=TRUE}
# head(dayList)
lapply(dayList, dim)
```

General Class Information

There are two useful functions associated with practically all R classes, which relate to logically checking the underlying class (`is.CLASS_()`) and coercing between classes (`as.CLASS_()`).

We saw some examples of coercion in the past, like `as.numeric()` and `as.character()` regarding the `factor` class and also `as.Date()` for the `date` class.

#part2
##Agenda

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

##More on data frames
```{r}
library(MASS)
head(Cars93, 3)
```

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

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

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

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

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

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


##Changing levels of a factor
```{r}
manufacturer <- Cars93$Manufacturer
head(manufacturer, 10)
```

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

```{r}
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)
```

##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

```{r}
survey <- read.table("survey_data.csv", header=TRUE, sep=",")
survey <- transform(survey, Program=as.numeric(Program))
head(survey)
```

##Example continued

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

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


##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

```{r}
table(Cars93$AirBags)
```
###
```{r}
table(Cars93$Origin)
```

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

- 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

```{r}
head(Cars93[c("AirBags", "Origin")], 3)
table(Cars93[c("AirBags", "Origin")])
```

##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.

```{r}
my.vector.1 <- c("Michael", 165, TRUE) # (name, weight, is.male)
my.vector.1
typeof(my.vector.1)  # All the elements are now character strings!
```

##Lists vs. vectors

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

- 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

```{r}
my.list <- list("Michael", 165, TRUE)
my.list
sapply(my.list, typeof)
~~~~

```

##Named elements
```{r}
patient.1 <- list(name="Michael", weight=165, is.male=TRUE)
patient.1
```

##Referencing elements of a list (similar to data frames)
```{r}
patient.1$name # Get "name" element (returns a string)
patient.1[["name"]] # Get "name" element (returns a string)
patient.1["name"] # Get "name" slice (returns a sub-list)
c(typeof(patient.1$name), typeof(patient.1["name"]))
```


##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

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

- `x` is the **argument** or **input**

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

```{r}
addOne(12)
```

##More interesting example

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

```{r}

```
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)
```

- 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.

```{r}
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
```{r}
createPatientRecord("Michael Smith", 150, 12 * 6 + 1)
```

##Another example: 3 number summary
- Calculate mean, median and standard deviation
```{r}
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)
```

##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
```{r}
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)
```

##`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

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

addOne(12)
```

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