Manipulate data frame

David Li

Revisit Data frame

- A "data matrix" or a "data set"
  - it likes a matrix (rectangular grid)
  - But unlike matrix, different columns can be of different types
  - Row names have to be unique
- >alphabet<-data.frame(index=1:26, symbol=LETTERS)
- read.table() reads file into a *data frame*
- Access var in a dataset: $, attach(), with()

```
>library(ISwR)  #load the package that provides thuesen data
>data(thuesen)
>names(thuesen)  #variable names
> blood.glucose # not visible
>length(thuesen$blood.glucose)
>with(thuesen, range(blood.glucose))
```

```
>attach(thuesen)
>range(blood.glucose)
>detach(thuesen)
```

Manipulate data frame

- Indexing of data frames is the same as that of vector and matrix >energy[energy$stature== "lean",]
- Sorting rows by order()

  >energy[order(energy$expend),]

  >energy[with(energy, order(stature, expend)),]
- Selecting subsets of data by subset()

  >subset(energy, stature=="lean" & expend>8)
- Splitting data

  >split(energy$expend, energy$stature)

Retrieve data in a cell

- ```
  > mtcars
                  mpg cyl disp  hp drat   wt ...
  Mazda RX4      21.0   6  160 110 3.90 2.62 ...
  Mazda RX4 Wag  21.0   6  160 110 3.90 2.88 ...
  Datsun 710     22.8   4  108  93 3.85 2.32 ...
  ```

- Access by index
  - ```
    > mtcars[1, 2]
    [1] 6
    ```
- Access by name
  - ```
    > mtcars["Mazda RX4", "cyl"]
    [1] 6
    ```
    ```

Data frame is a list of vectors with same length

- ```
  > mtcars
                  mpg cyl disp  hp drat   wt ...
  Mazda RX4      21.0   6  160 110 3.90 2.62 ...
  Mazda RX4 Wag  21.0   6  160 110 3.90 2.88 ...
  Datsun 710     22.8   4  108  93 3.85 2.32 ...
  ```

> typeof(mtcars)
[1] "list"
> class(mtcars)
[1] "data.frame"

- reference a data frame column with the *double square bracket* "[[]]" operator.
  - > `mtcars[[1]]`
    [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 ...
- retrieve the same column vector by its name.
  - > `mtcars[["mpg"]]`
    [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 ...
- retrieve with the "$" operator in lieu of the double square bracket operator.
  - > `mtcars$mpg`
    [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 ...
- use the single square bracket "[]" operator
  - > `mtcars[, 'mpg']`
    [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 ...
  ```

Column Slicing

- ```
  > mtcars
                  mpg cyl disp  hp drat   wt ...
  Mazda RX4      21.0   6  160 110 3.90 2.62 ...
  Mazda RX4 Wag  21.0   6  160 110 3.90 2.88 ...
  Datsun 710     22.8   4  108  93 3.85 2.32 ...
  ```

- Numeric Indexing
  - ```
    > mtcars[1]
                        mpg
    Mazda RX4          21.0
    Mazda RX4 Wag      21.0
    Datsun 710         22.8
                        .............
    ```

- Name Indexing
  - ```
    > mtcars["mpg"]
                        mpg
    Mazda RX4          21.0
    Mazda RX4 Wag      21.0
    Datsun 710         22.8
                        .............
    ```
  - ```
    > mtcars[c("mpg", "hp")]
                        mpg  hp
    Mazda RX4          21.0 110
    Mazda RX4 Wag      21.0 110
    Datsun 710         22.8  93
                        .............
    ```

Row Slicing

- ```
  > mtcars
                  mpg cyl disp  hp drat   wt ...
  Mazda RX4      21.0   6  160 110 3.90 2.62 ...
  Mazda RX4 Wag 21.0   6  160 110 3.90 2.88 ...
  Datsun 710     22.8   4  108  93 3.85 2.32 ...
  ```
- Numeric Indexing
  - ```
    > mtcars[2,]
                  mpg cyl disp hp drat wt qsec vs am gear carb
    Mazda RX4 Wag 21 6 160 110 3.9 2.875 17.02 0 1 4 4
    ```
  - ```
    > mtcars[c(1,3),]
                  mpg cyl disp hp drat wt qsec vs am gear carb
      Mazda RX4 21.0 6 160 110 3.90 2.62 16.46 0 1 4 4
      Datsun 710 22.8 4 108 93 3.85 2.32 18.61 1 1 4 1
    ```
- Name Indexing
  - ```
    > mtcars["Mazda RX4 Wag",]
                  mpg cyl disp hp drat wt qsec vs am gear carb
    Mazda RX4 Wag 21 6 160 110 3.9 2.875 17.02 0 1 4 4
    ```
  - ```
    > mtcars[c("Mazda RX4", "Datsun 710"),]
                  mpg cyl disp hp drat wt qsec vs am gear carb
      Mazda RX4 21.0 6 160 110 3.90 2.62 16.46 0 1 4 4
      Datsun 710 22.8 4 108 93 3.85 2.32 18.61 1 1 4 1
    ```
    ```

Row Slicing

- ```
  > mtcars
                  mpg cyl disp  hp drat   wt ...
  Mazda RX4      21.0   6  160 110 3.90 2.62 ...
  Mazda RX4 Wag 21.0   6  160 110 3.90 2.88 ...
  Datsun 710     22.8   4  108  93 3.85 2.32 ...
  ```
- Logical Indexing
  - ```
    > low_mpg = mtcars$mpg < 22
    > low_mpg
    [1] TRUE TRUE FALSE...
    > mtcars[low_mpg,]
                    mpg cyl disp hp drat wt qsec vs am gear carb
      Mazda RX4     21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4
      Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4
    ```
  - ```
    > mtcars[low_mpg,]$wt
    [1] 2.620 2.875 ...
    ```
    ```

Introduction to dplyr

- The dplyr package was developed by Hadley Wickham of RStudio and is an optimized and distilled version of his plyr package.
- Provides a "grammar" (in particular, verbs) for data manipulation and for operating on data frames.
- Provides an abstraction for data manipulation that previously did not exist
- dplyr functions are **very** fast, as many key operations are coded in C++.

Important dplyr verbs

dplyr verbs	Description
select()	select columns
filter()	filter rows
arrange()	re-order or arrange rows
mutate()	create new columns
summarise()	summarise values
group_by()	allows for group operations in the "split-apply-combine" concept

Common dplyr Function Properties

- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the $ operator (just use the column names).
- The return result of a function is a new data frame
- Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be tidy. In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.

Example Data

- #install.packages("dplyr") install it
- library("dplyr")
- msleep <- read.csv(“msleep_ggplot2.csv")

```
> str(msleep)
'data.frame':   83 obs. of  11 variables:
 $ name        : Factor w/ 83 levels "African elephant",..: 12 57 52 36 17 77 55 81 21 67 ...
 $ genus       : Factor w/ 77 levels "Acinonyx","Aotus",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ vore        : Factor w/ 4 levels "carni","herbi",..: 1 4 2 4 2 2 1 NA 1 2 ...
 $ order       : Factor w/ 19 levels "Afrosoricida",..: 3 15 17 19 2 14 3 17 3 2 ...
 $ conservation: Factor w/ 6 levels "cd","domesticated",..: 4 NA 5 4 2 NA 6 NA 2 4 ...
 $ sleep_total : num  12.1 17 14.4 14.9 4 14.4 8.7 7 10.1 3 ...
 $ sleep_rem   : num  NA 1.8 2.4 2.3 0.7 2.2 1.4 NA 2.9 NA ...
 $ sleep_cycle : num  NA NA NA 0.133 0.667 ...
 $ awake       : num  11.9 7 9.6 9.1 20 9.6 15.3 17 13.9 21 ...
 $ brainwt     : num  NA 0.0155 NA 0.00029 0.423 NA NA NA 0.07 0.0982 ...
 $ bodywt      : num  50 0.48 1.35 0.019 600 ...
```

Data Description

column name	Description
name	common name
genus	taxonomic rank
vore	carnivore, omnivore or herbivore?
order	taxonomic rank
conservation	the conservation status of the mammal
sleep_total	total amount of sleep, in hours
sleep_rem	rem sleep, in hours
sleep_cycle	length of sleep cycle, in hours
awake	amount of time spent awake, in hours
brainwt	brain weight in kilograms
bodywt	body weight in kilograms

dplyr verbs in action

- select(): selects columns

```
#Select a set of columns: the name and the sleep_total columns.
>sleepData <- select(msleep, name, sleep_total)
>head(sleepData)

# Excluding a specific column, use the "- "
>head(select(msleep, -name))

# select a range of columns by name, use the ":"
>head(select(msleep, name:order))

#select all columns that start with the character string "sl" , # use the function starts_with()
> head(select(msleep, starts_with("sl"))) #does starts_with return a logical vector?
```

Some additional options to select columns

- ends_with() = Select columns that end with a character string
- contains() = Select columns that contain a character string
- matches() = Select columns that match a regular expression
- one_of() = Select columns names that are from a group of names

```
> head(select(msleep, ends_with("e")))
> head(select(msleep, matches("e$")))
> head(select(msleep, contains("_")))
> head(select(msleep, matches("_")))
> head(select(msleep, one_of("name", "order")))
```

Selecting rows using filter()

- filter(mtcars, cyl == 8)
- filter(mtcars, cyl < 6)

Multiple criteria

- filter(mtcars, cyl < 6 & vs == 1)
- filter(mtcars, cyl < 6 | vs == 1)

Multiple arguments are equivalent to and

- filter(mtcars, cyl < 6, vs == 1) #see … in ?filter

Difference between subset and filter

- They produce the same result
- subset is that it is part of base R and doesn't require any additional packages
- filter is a function of dplyr package
- subset is faster with small sample size
- filter is faster with large sample size (> 15000 records)

arrange()

- The arrange() function is used to reorder rows of a data frame according to one/some of the variables/-columns

```
>arrange(mtcars, cyl, disp)
>arrange(mtcars, desc(disp))
```

rename()

- Renaming a variable in a data frame in R is surprisingly hard to do! The rename() function is designed to make this process easier.
- rename(data, new.name=old.name)
- Example
  - rename(mtcars, Weight=wt)
  - rename(mtcars, wt=Weight) #error

mutate() / transmute()

- mutate() computes transformations of variables in a data frame
- Often, you want to create **new** variables that are derived from existing variables.

```
> head(mutate(msleep, sleep_total_min = sleep_total * 60))
```

- transmute() function, the same as mutate() but then *drops all non-transformed variables*.

```
> head(transmute(msleep, sleep_total_min = sleep_total * 60))
```

group_by()

- generate summary statistics from the data frame within strata defined by a variable.

```
>by_cyl <- group_by(mtcars, cyl)
>summarise(by_cyl, mean(disp), mean(hp))

>by_vore <- group_by(msleep, vore)
> summarise(by_vore, total=mean(sleep_total),
  avg_sleep_rem=mean(sleep_rem, na.rm=T))
```

Pipe operator %>%

- Passing the result of one step as input for the next step in a sequence of operations.
- Easy to read
- Syntax
  - lhs %>% rhs # pipe syntax for rhs(lhs)
  - lhs %>% rhs(a = 1) # pipe syntax for rhs(lhs, a = 1)
  - lhs %>% rhs(a = 1, b = .) # pipe syntax for rhs(a = 1, b = lhs)

```
> third(second(first(x)))
```

vs

```
> first(x) %>% second %>% third
```

%>% example

- select three columns from msleep, arrange the rows by the taxonomic order and then arrange the rows by sleep_total. Finally show the head of the final data frame

```
> msleep %>% select(name, order, sleep_total) %>% arrange(order, sleep_total) %>% head
```

- Same as above, except here we filter the rows for mammals that sleep for 16 or more hours instead of showing the head of the final data frame

```
> msleep %>% select(name, order, sleep_total) %>% arrange(order, sleep_total) %>% filter(sleep_total >= 16)
```

```
> msleep %>% group_by(order) %>% summarise(avg_sleep = mean(sleep_total), min_sleep = min(sleep_total), max_sleep = max(sleep_total), total = n()) # n() (returns the length of vector)
```

Summary

- The dplyr package provides a concise set of operations for managing data frames.
- With these functions we can do a number of complex operations in just a few lines of code
- In particular, we can often conduct the beginnings of an exploratory analysis with the powerful combination of group_by() and summarize().
- dplyr can work with other data frame "backends" such as SQL databases. There is an SQL interface for relational databases via the DBI package
- dplyr can be integrated with the data.table package for large fast tables
- Both **simplify** and **speed up** your data frame management code.

reshape2 package

- Reshape2 is a reboot of the reshape package, also developed by Hadley Wickham
- It makes it easy to transform data between wide and long formats.
- Much more focused and much much faster.
- install.packages("reshape2")

What makes data wide or long?

- Wide data has a column for each variable.

Wide-format

```
#    ozone   wind   temp
# 1 23.62 11.623  65.55
# 2 29.44 10.267  79.10
# 3 59.12   8.942  83.90
# 4 59.96   8.794  83.97
```

long-format

```
#    variable  value
# 1     ozone 23.615
# 2     ozone 29.444
# 3     ozone 59.115
# 4     ozone 59.962
# 5      wind 11.623
# 6      wind 10.267
# 7      wind  8.942
# 8      wind  8.794
# 9      temp 65.548
# 10     temp 79.100
# 11     temp 83.903
# 12     temp 83.968
```

- Long-format data has a column for possible variable types and a column for the values of those variables.
- Long-format data isn’t necessarily only two columns.
- More commonly used than wide-format: ggplot2

Two major functions

- melt: takes wide-format data and melts it into long-format data.
- cast: takes long-format data and casts it into wide-format data.

Think of working with metal: if you melt metal, it drips and becomes long. If you cast it into a mould, it becomes wide.

melt

```
names(airquality) <- tolower(names(airquality))
head(airquality)
```

•By default, melt has assumed that all columns with numeric values are variables with values.

```
aql <- melt(airquality)
head(aql)
tail(aql)
```

melt

- Specify "ID variables", the variables that identify individual rows of data.

```
aql <- melt(airquality, id.vars = c("month", "day"))
head(aql)
subset(airquality, month==5 & day==1)
subset(aql, month==5 & day==1)
```

- Set column names

```
aql <- melt(airquality, id.vars = c("month", "day"),
  variable.name = "climate_variable",
  value.name = "climate_value")
head(aql)
```

Ex: aql2 <- melt(airquality, id.vars = c("month", "day", "ozone"))

cast

- dcast: work with data.frame objects; acast: return a vector, matrix, or array
- dcast uses a formula to describe the shape of the data.
- The arguments on the left refer to the ID variables and the arguments on the right refer to the measured variables.
- Coming up with the right formula can take some trial and error at first. So, if you're stuck don't feel bad about just experimenting with formulas.

cast

```
aql <- melt(airquality, id.vars = c("month", "day"))
aqw <- dcast(aql, month + day ~ variable)
head(aqw)
head(airquality) # original data
```

id.vars is like the composite key in database

dcast formula `dcast(aql, month + day ~ variable, value.var = "value")`

ID variables (left side of formula)	Variable to swing into column names (right side of formula)	Values (value.var)

Long-format data

month	day	variable	value
5	1	ozone	41
5	2	ozone	36
5	3	ozone	12
5	4	ozone	18
5	5	ozone	NA
5	6	ozone	28

Wide-format data

month	day	ozone	solar.r	wind	temp
5	1	41	190	7.4	67
5	2	36	118	8.0	72
5	3	12	149	12.6	74
5	4	18	313	11.5	62
5	5	NA	NA	14.3	56
5	6	28	NA	14.9	66

More than one value per data cell

- One confusing "mistake" you might make is casting a dataset in which there is more than one value per data cell.
- Example:

dcast(aql, month ~ variable)

When you run this in R, you'll notice the warning message:

Aggregation function missing: defaulting to length

Aggregate the data

- dcast(aql, month ~ variable, fun.aggregate = mean, na.rm = TRUE)
- Unlike melt, there are some other fancy things you can do with dcast that I’m not covering here. It’s worth reading the help file ?dcast. For example, you can compute summaries for rows and columns, subset the columns, and fill in missing cells in one call to dcast.

Other resources

- http://genomicsclass.github.io/book/pages/dplyr_tutorial.html
- https://github.com/hadley/reshape
- http://seananderson.ca/2013/10/19/reshape.html
- http://had.co.nz/reshape/
- help(package = "reshape2")
- Reshaping data with the reshape package. 21(12):1–20. Wickham, H. (2007). http://www.jstatsoft.org/v21/i12 (But note that the paper is written for the reshape package not the reshape2 package.)

Introduction to Functional programming in R

David Li

Functions

Functions in R

- A core activity of an R programmer.
- "user" → developer
- When to write a function
  - Encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions.
  - Code must be shared with others or the public
- Create an interface to the code: via a set of parameters.
- This interface provides an abstraction of the code to potential users.
  - Ex: sort()

Your First Function

```
f <- function() {
   ## This is an empty function
}
## Functions have their own class
class(f)
# Execute this function
 f()
```

```
#more fun
f <- function() {
   cat("Hello, world!\n")
}
f()
```

```
#with a parameter
f <- function(num) {
 for(i in seq_len(num)) {
  cat("Hello, world!\n")
  }
}
f(3)
```

```
# with return value
f <- function(num) {
  hello <- "Hello, world!\n"
   for(i in seq_len(num)) {
   cat(hello)
   }
   chars <- nchar(hello) * num
   chars  # logical last expression
returned
}
meaningoflife = f(3)
```

#return the very last expression that is evaluated.

Default value

```
f()
f <- function(num = 1) {
  hello <- "Hello, world!\n"
  for(i in seq_len(num)) {
    cat(hello)
  }
  chars <- nchar(hello) * num
  chars
}
f() ## Use default value for 'num'

f(2)
f(num=2) #specified using argument name
```

So far, we have written a function that

- has one *formal argument* named num with a *default value* of 1.
- prints the message "Hello, world!" to the console a number of times indicated by the argument num
- *returns* the number of characters printed to the console

Argument Matching

- R functions arguments can be matched *positionally* or by name.
- Positional matching just means that R assigns the first value to the first argument, the second value to second argument, etc.

```
> str(rnorm)
function (n, mean = 0, sd = 1)
> set.seed(0)
> mydata <- rnorm(100, 2, 1) ## Generate some data
```

100 is assigned to the n argument, 2 is assigned to the mean argument, and 1 is assigned to the sd argument, all by positional matching.

Specifying arguments by name

- Order doesn’ t matter then

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

Here, the mydata object is assigned to the x argument, because it’s the only argument not yet specified.

- Function arguments can also be *partially* matched

- The order of operations when given an argument
  1. Check for exact match for a named argument
  2. Check for a partial match
  3. Check for a positional match

Example

```
> 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
```

The following two calls are equivalent.

```
set.seed(0)
mydata =  data.frame(y=rnorm(20), x=rnorm(20))
lm(data = mydata, y ~ x, model = FALSE, 1:20)
lm(y ~ x, mydata, 1:20, model = FALSE)
```

Lazy Evaluation

- Arguments to functions are evaluated *lazily*, so they are evaluated only as needed in the body of the function.

```
> f <- function(a, b) {
  a^2
 }
> f(2)  #it works and doesn't report error
```

```
> f <- function(a, b) {
  print(a)
  print(b)
}
> f(45)  # error, argument  "b"  is missing with no default
```

The ... Argument (like **args in python function)

- A special argument in R
- Indicate a variable number of arguments that are usually passed on to other functions.

```
myplot <- function(x, y, type = "l", ...) {
  plot(x, y, type = type, ...) ## Pass '...' to 'plot' function
}
```

- The ... argument is necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL
> args(cat)
function (..., file = "", sep = " ", fill = FALSE, labels = NULL, append = FALSE)
NULL
```

Arguments Coming After the ... Argument

- One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched or matched positionally.

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL
paste("a", "b", sep = "+")
#full arg name matches  "sep"  in the function
paste("a", "b", se = "+")
#partial arg name becomes part of ...
```

Summary

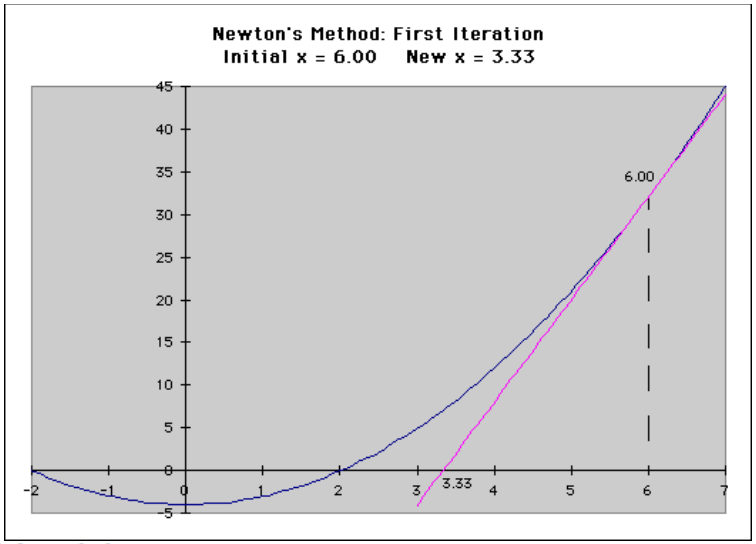
- Functions can be defined using the function() directive and are assigned to R objects just like any other R object
- Functions have can be defined with named arguments; these function arguments can have default values
- Functions arguments can be specified by name or by position in the argument list
- Functions always return the last expression evaluated in the function body
- A variable number of arguments can be specified using the special ... argument in a function definition.

Example: Newton-Rapson method to find a square root of an integer number

It is an iterative number method. This type of methods create a succecion $(x_0, x_1 \cdots x_n)$.
With some initial conditions given a root of the fucntion f, α

$$f(\alpha) = 0$$

$$\lim_{n->inf} \alpha - x_n = 0$$



More exactly newton-rapson follow the next schema:

$$x_{n+1} = x_n - \frac{f(x)}{f'(x_n)}$$

For finding a square root of an integer (t) you can do this trick.

$$f(x) = x^2 - t$$

then in our case:

$$x_{n+1} = x_n - \frac{(x_n)^2 - t}{2x_n}$$

where f'(x) = 2x.

Functional Programming

Looping on the Command Line

- apply(): Apply a function over the margins of an array
- lapply(): Loop over a list and evaluate a function on each element
- sapply(): Same as lapply but try to simplify the result
- mapply(): Multivariate version of lapply
- tapply(): Apply a function over subsets of a vector

apply()

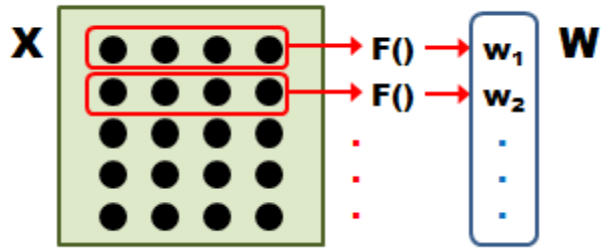
- Evaluate a function (often an anonymous one) over the margins of an array.
- Most often, apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). Also, applicable to general arrays

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

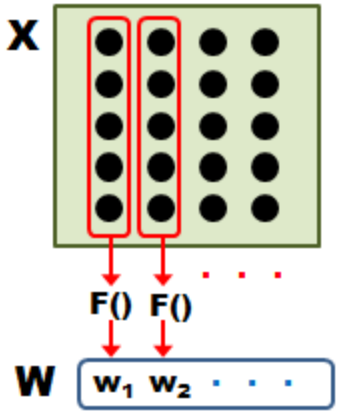
- The arguments to apply() are
  - X is an array (matrix is just a 2D array)
  - MARGIN is an integer vector indicating which margins should be "retained".
  - FUN is a function to be applied
  - ... is for other arguments to be passed to FUN

apply()

W <- apply(X, 1, F)



W <- apply(X, 2, F)



apply()

- Examples

```
> set.seed(0)
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean) ## Take the mean of each column
> apply(x, 1, sum) ## Take the sum of each row
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
```

- Shortcuts
  - rowSums = apply(x, 1, sum)
  - rowMeans = apply(x, 1, mean)
  - colSums = apply(x, 2, sum)
  - colMeans = apply(x, 2, mean)

lapply()

- The lapply() function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)
3. and returns a list (the l is for "list").

- This function takes three arguments
  - (1) a list X, If X is not a list, it will be coerced to a list using as.list().
  - (2) a function (or the name of a function) FUN;
  - (3) other arguments via its ... argument.

lapply()

- the actual looping is done internally in C code for efficiency reasons.

```
> lapply
function (X, FUN, ...)
{
FUN <- match.fun(FUN)
if (!is.vector(X) || is.object(X))
X <- as.list(X)
.Internal(lapply(X, FUN))
}
<bytecode: 0x7fcc8388f758>
<environment:
namespace:base>
```

lapply()

- Example 1

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
```

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
```

- Example 2

```
> x <- 1:4
> lapply(x, runif)
```

When you pass a function to lapply(), lapply() takes elements of the list and passes them as the *first argument* of the function you are applying.

lapply()

- the ... Argument

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
```

- *anonymous* functions.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> lapply(x, function(elt) { elt[,1] })
```

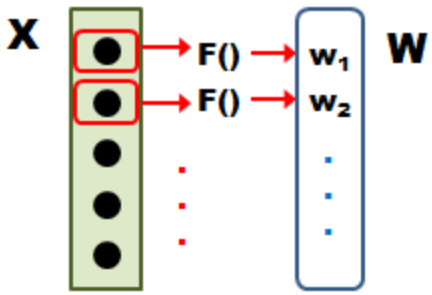
V.S.

```
> f <- function(elt) {
    elt[, 1]
  }
> lapply(x, f)
```

sapply()

- The sapply() function behaves similarly to lapply(); the only real difference is in the return value.

W <- lapply(X, F)
W <- sapply(X, F)



- sapply() will try to simplify the result of lapply() if possible. Essentially, sapply() calls lapply() on its input and then applies the following algorithm
  - If the result is a list where every element is length 1, then a vector is returned
  - If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
  - If it can't figure things out, a list is returned

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)

> sapply(x, mean)
```

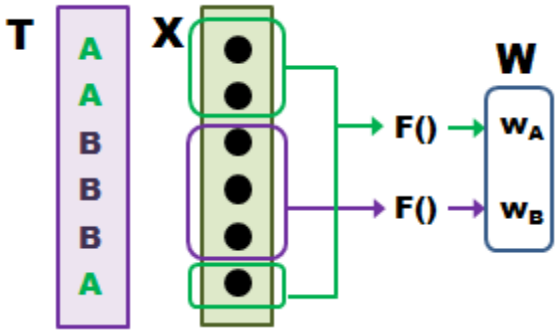
split()

- The combination of split() and a function like lapply() or sapply() is a common paradigm in R.

```
> library(datasets)
> head(airquality)

> s <- split(airquality, airquality$Month)
> str(s)

> lapply(s, function(x) {
  colMeans(x[, c("Ozone", "Solar.R", "Wind")])
  })

> sapply(s, function(x) {
   colMeans(x[, c("Ozone", "Solar.R", "Wind")])
 })

> sapply(s, function(x) {
  colMeans(x[, c("Ozone", "Solar.R", "Wind")],na.rm = TRUE)
 })
```

tapply

- tapply() is used to apply a function over subsets of a vector. It can be thought of as a combination of split() and sapply() for vectors only.



W <- tapply(X, T, F)

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

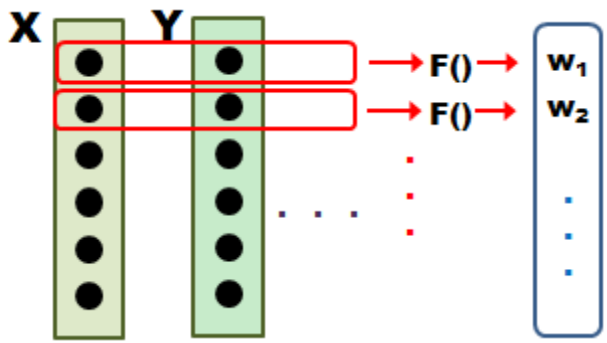
- The arguments to tapply() are as follows:
  - X is a vector
  - INDEX is a factor or a list of factors (or else they are coerced to factors)
  - FUN is a function to be applied
  - … contains other arguments to be passed FUN
  - simplify, should we simplify the result?

tapply

```
> ## Simulate some data
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> ## Define some groups with a factor variable
> f <- gl(3, 10)
> f
[1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)


> tapply(x, f, mean, simplify = FALSE)
> tapply(x, f, mean, simp = FALSE) #will it return the same? why
# when returning >1 value. tapply() will not simplify the result and
# will return a list.
> tapply(x, f, range)
```

mapply()

- A multivariate apply of sorts which applies a function in parallel over a set of arguments.
- Recall that lapply() and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what mapply() is for.



W <- mapply(F, X, Y, ...)

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES =
TRUE)
```

The arguments to mapply() are

- FUN is a function to apply
- ... contains R objects to apply over
- MoreArgs is a list of other arguments to FUN.
- SIMPLIFY indicates whether the result should be simplified

mapply()

list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))

```
> mapply(rep, 1:4, 4:1)
> noise <- function(n, mean, sd) {
  rnorm(n, mean, sd)
 }
> ## Simulate 5 random numbers
> noise(5, 1, 2)
[1] -0.5196913 3.2979182 -0.6849525 1.7828267 2.7827545
>
> ## This only simulates 1 set of numbers, not 5
> noise(1:5, 1:5, 2)
[1] -1.670517 2.796247 2.776826 5.351488 3.422804
> mapply(noise, 1:5, 1:5, 2)      ==      list(noise(1, 1, 2), noise(2, 2, 2),
                                           noise(3, 3, 2), noise(4, 4, 2),
                                           noise(5, 5, 2))
```

Vectorizing a Function

- The mapply() function can be used to automatically "vectorize" a function: take a function that typically only takes single arguments and create a new function that can take vector arguments.

```
> sumsq <- function(mu, sigma, x) {
   sum(((x - mu) / sigma)^2)
  }
> x <- rnorm(100) ## Generate some data
> sumsq(1:10, 1:10, x) ## This is not what we want
[1] 110.2594
```

However, we can do what we want to do by using mapply().

```
> mapply(sumsq, 1:10, 1:10, MoreArgs = list(x = x))
```

Vectorize()

- It can automatically create a vectorized version of your function.
- Example: create a vsumsq() function that is fully vectorized as follows.

```
> vsumsq <- Vectorize(sumsq, c("mu", "sigma"))
> vsumsq(1:10, 1:10, x)
[1] 196.2289 121.4765 108.3981 104.0788 102.1975 101.2393 100.6998
[8] 100.3745 100.1685 100.0332
```

Summary

- The loop functions in R are very powerful because they allow you to conduct a series of operations on data using a compact form
- The operation of a loop function involves iterating over an R object (e.g. a list or vector or matrix), applying a function to each element of the object, and the collating the results and returning the collated results.
- Loop functions make heavy use of anonymous functions, which exist for the life of the loop function but are not stored anywhere
- The split() function can be used to divide an R object in to subsets determined by another variable which can subsequently be looped over using loop functions.

More information

- Further Readings
  - http://adv-r.had.co.nz/Functionals.html
  - https://towardsdatascience.com/functional-programming-in-r-with-purrr-469e597d0229
  - Speed up the loop operation in R
    - http://stackoverflow.com/questions/2908822/speed-up-the-loop-operation-in-r
  - Is R's apply family more than syntactic sugar?
    - http://stackoverflow.com/questions/2275896/is-rs-apply-family-more-than-syntactic-sugar

Lab exercise: Bond Volume Analysis

Data Preparation

Download the csv file and load into data frame.

https://njit.instructure.com/files/483333/download?download_frd=1&verifier=In8pFaKc70FCfXfIQJ0BkKp9RxuJ4fHatnZRo91X

Write R code to answer the following questions

1. What's the total amount issued in each currency? Sort the result by total amount with the largest amount on the top.
2. How many bonds in each currency?
3. How many bonds issued by each company?
4. What's the total amount issued in each currency by each company? Result should look like

   Name | Total Amount Issued | Currency
5. Change all currencies to columns and the total amount issued to the value under the currency column so that the data frame looks like

Name	EUR	CHF	SEK
Telia Co	500000000	NA	NA