Data Structures

Contents

Tutorial 1 **2**

Preamble . . . 2
Questions . . . 2
Introductory Questions . . . 3
1. What is meant by Working Directory in RStudio? . . . 3
2. Why is it important to know the working directory? . . . 3
3. How do we set the working directory? . . . 3
4. Name some of the frequently used data types in R? . . . 4
5. Assignment Characters . . . 5

Working with Vectors and Lists . . . 5
6. Write an example of a logical vector. . . . 5
7. Write an example of a numeric vector. . . . 6
8. Write an example of a character vector. . . . 6
9. Write an example of an integer vector. . . . 6
10. Convert the following integer vector to character vector. . . . 6
11. Convert the following integer vector to logical vector. . . . 6
12. Write an example of a list. . . . 7
13. Add 5 to each element of the vector. . . . 8
14. Sort the elements of the vector. . . . 8
15. Get the first three elements of the vector (sub-setting vectors). . . . 9
15. Get all elements except 3rd. . . . 9
16. Get all elements except 1st and 3rd. . . . 9
17. Get all elements greater than 3. . . . 9
18. Get all elements greater than 3 but less than 7. . . . 10
19. Get all indices of elements greater than 3 but less than 7. . . . 10
20. Get all sum of elements greater than 3 but less than 7. . . . 11
21. Add one to all numbers not divisible by 3. . . . 11
21. Replace those numbers not divisible by 3 by NA (Not a number). . . . 11
22. Find the count of numbers in the vector. . . . 11
22. Print each number on a new line . . . 12
23. Given the character vector, find its factors . . . 15
24. Given the character vector, find the structure of how it is stored . . . 16
26. Given the character vector, change the annotations for the levels. . . . 16
27. Forcing the levels to be in a specific order. . . . 16
28. Given the character vector find the frequency of each level. . . . 17

Using the Sample Function . . . 17
29. Given the character vector what does the following code do? . . . 17

Working with numeric data . . . 18

The Data Frame (The R equivalent of the spreadsheet) . . . 19
30. Given the following data frame, find just the names of the people. . . . 20

31. Given the following data frame, get all the details of the 1st and 20
32. Given the following data frame, get all the details of the first three persons. 20
33. Given the following data frame, get all the details of those 20
34. Given the following data frame, add another column representing the 21
34. Given the following data frame, add two more rows of data 21
36. Assuming you have file named "BeanData.csv" in your working 22
37. Assuming you have initialized beans by importing from an Excel 24
38. Observe the first 6 rows of data in *beans*. 24
39. Use the *apply* function to find the means of the last three 24
40. Observe that the beans data has trt representing certain grouping. 25
41. Using `tapply` for multiple 26
44. Custom Functions can be used with `tapply()` 30
47. Normalize a Vector 31
48. Normalize a Matrix 32
References 32

Tutorial 1

Week 1 for Sydney; Week 2 for Western Sydney Material Covered 2020-March-13

Preamble

```r
# Preamble
## Install Pacman
load.pac <- function() {
  if (require("pacman")) {
    library(pacman)
  }else{
    install.packages("pacman")
    library(pacman)
  }
  pacman::p_load(xts, sp, gstat, ggplot2, rmarkdown, reshape2, ggmap,
                 parallel, dplyr, plotly, tidyverse, reticulate, UsingR, Rmpfr,
                 swirl, corrplot, gridExtra, mise, latex2exp)
}

load.pac()
mise()
```

```r
set.seed(9823)
```

Questions

Introductory Questions

1. What is meant by Working Directory in RStudio?

The working directory in ***R*** is the base directory, it set's the root/home directory of the script; the practical idea being that resources read to, and written from, the drive relative to the working directory.

2. Why is it important to know the working directory?

It is important to know the working directory because it is relative to this location that resources will be read from and written to. Generally resources will be manipulated from other software (OS/bash/python etc.) and so a relative working directory is necessary to work with those resources from ***R***.

3. How do we set the working directory?

There's a few different ways to set the working directory depending on what you want to do.

Default Working directory

By default the working directory is set (I think) to the home directory `~` in `*nix` and to `C:/Users/Username` in *Windows*.

Set Working Directory to Directory

A specific directory can be specified by using the following command:

```
setwd("~/Notes/DataSci/Social_Web_Analytics")
```

```
print("hello World")
```

```
## [1] "hello World"
```

```
# cat("/014")
```

Set Working Directory to File Location

Reading `parent.frame(2)` Generally it's helpful to set the working directory to the location of the script, regardless of whether or not the script is moved, using ***R*** this can be acheived thusly:

```
this.dir <- dirname(parent.frame(2)$ofile)
setwd(this.dir)
rm(this.dir)
```

This method has the disadvantage of breaking ***R*** *Studio*'s `KnitR` (and may interfere with `spin`) so if you're going to use that the below method, using `rstudioapi` is better.

Using `rstudioapi` When using ***R*** *Studio* the rstudioapi may be leveraged in order to set the working directory, the disadvantage to this though is that ***R*** *Studio* must be installed and loading the `api` can be slow if the script is supposed to be relatively quick.

```
#Test and install pacman package
if (require('pacman')) {
  library('pacman')
} else{
  install.packages('pacman')
  library('pacman')
}

#Use Pacman to install other useful packages
pacman::p_load(ggplot2, rmarkdown, dplyr, plotly, rstudioapi, installr, reshape2)

#Use the Rstudio API to get the working directory
current_path <- getActiveDocumentContext()$path

# Set the Working Directory
setwd(dirname(current_path))
```

R Markdown If you're using ***R*** *Markdown* the working directory is just the location of the `.rmd` file, which is fairly logical for that type of a file.

HTML Notebooks / HTML Document

HTML Notebooks are rendered very quickly with a default style automatically upon saving, it's close to live, but it's generally not as good as *iamcco's Preview* or *Zettlr* or *MarkText*.

A HTML Document will be knitted and every code block will be evaluated each time, it's too slow to be practical.

4. Name some of the frequently used data types in R?

`numeric` Data

This is Numeric Data, it is a floating point number equivalent to `float` or `double` in Java.

`character` Data

The character data is any type of string data, unlike Java this is no data type for only one character, in ***R*** a string and character are both considered to be of the `character` data type.

Ans: numeric, character, logical, complex, factors, matrices, arrays, and lists

5. Assignment Characters

What is the difference between?

```
\> a=2 \#or a \<-2 \#or 2 -\> a
```

And

```
\> A=3 \# or A \<- 3 \#or 3 -\> A
```

Ans: _R_** is case sensitive: a and A are two different variables and are assigned values 2 and 3 respectively. Assignment operations may be either done using = or <- or -> operators. A variable is an object in r.**

The <- character is recommended by various style guides, there may also be a slight difference.

Working with Vectors and Lists

The most basic object is a vector. A vector can only contain objects of the same type (also known as class). BUT: The one exception is a *list*, which is represented as a vector but can contain objects of different classes

6. Write an example of a logical vector.

Try to be aware that any non-zero value tends to be *truthy*

```
logical_vector <- c(TRUE, FALSE, TRUE)
(numbers <- (sample(0:3, size = 90, replace = TRUE)))
```

```
##  [1] 1 3 0 1 2 1 3 0 3 1 3 1 1 2 3 1 3 0 0 0 0 0 2 1 2 3 1 3 2 2 1 1 3 1 1 1 3 0
     1 3 1 2 0 1 2
## [46] 1 3 0 0 2 3 2 2 2 3 2 3 3 2 0 0 3 3 2 1 1 2 2 3 2 3 2 0 0 1 3 1 0 1 3 2 0 0
    2 2 2 1 0 1 1
```

```
logical_vector <- numbers %>% as.logical()
```

7. Write an example of a numeric vector.

```
(x <- c(1,3, 9, 9))
```

```
## [1] 1 3 9 9
```

8. Write an example of a character vector.

```
(x <- c("train", "furniture", "tracks"))
```

```
## [1] "train"     "furniture" "tracks"
```

9. Write an example of an integer vector.

It is necessary to specify integers as a vector

```
(integers <- as.integer(c(1,2,3)))
```

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

10. Convert the following integer vector to character vector.

```
x <- 1:3
class(x)
```

```
## [1] "integer"
```

```
y <- as.character(x)
```

11. Convert the following integer vector to logical vector.

```
x <- c(0, 2, -3)
y <- as.logical(x)
```

Note 0 is considered as FALSE, non-zero is TRUE.

Or well rather truthy

```
library(tidyverse)
nums <- ((sample(0:4, size = 9, replace = TRUE, prob = c(0.5, 0.1, 0.1, 0.1, 0.1))))

data.frame("Original Numbers" = nums, "Using as.logical" = as.logical(nums))
```

```
##   Original.Numbers Using.as.logical
## 1                0            FALSE
## 2                0            FALSE
## 3                0            FALSE
## 4                0            FALSE
## 5                0            FALSE
## 6                0            FALSE
## 7                0            FALSE
## 8                0            FALSE
## 9                0            FALSE
```

12. Write an example of a list.

```
( x <- list(3, "a", TRUE, 1+4i))
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

- `[[]]` are used to represent each element of a list.
- `x[[1]]` will be a vector of integer in which the first element will be 3
- `x[[2]]` will be a vector of characters in which the first element will be "a"
- `x[[3]]` will be a vector of logical values in which the first element will be TRUE
- `x[[4]]` will be a vector of complex values in which the first element will be 1+4i

```
(x <- list(c(1,3), c("a","d")))
```

```
## [[1]]
## [1] 1 3
##
```

```
## [[2]]
## [1] "a" "d"
```

```
print("Hello")
```

```
## [1] "Hello"
```

- `X[[1]]` will be a vector of integers with values 1, 3. X`[[1]][1]` will be 1 and `x[[1][2]` is 3
- `X[[2]]` will be a vector of chars. `X[[2]][1]` will be "a" and x[[2][2] will be "d"
- R can do many things easily with vectors: mathematical operations, sorting, and sub-setting.
  - I'm not sure how to do this with *TidyVerse* though.

13. Add 5 to each element of the vector.

```
(vec <- sample(1:4, size = 9, replace = TRUE))
```

```
## [1] 2 4 3 3 4 3 2 2 3
```

```
vec + 5
```

```
## [1] 7 9 8 8 9 8 7 7 8
```

14. Sort the elements of the vector.

```
y <- c(5, 1, 6)
sort(y)
```

```
## [1] 1 5 6
```

```
sort(y, decreasing = TRUE)
```

```
## [1] 6 5 1
```

15. Get the first three elements of the vector (sub-setting vectors).

```
(y <- c("a", "x", "y", "z"))
```

```
## [1] "a" "x" "y" "z"
```

```
y[1:3] #returns a vector with values c('a', 'x', 'y')
```

```
## [1] "a" "x" "y"
```

15. Get all elements except 3rd.

```
(y <- c("a", "x", "y", "z"))
```

```
## [1] "a" "x" "y" "z"
```

```
y[-3]
```

```
## [1] "a" "x" "z"
```

16. Get all elements except 1st and 3rd.

```
y <- c("a", "x", "y", "z")
y[-c(1,3)]
```

```
## [1] "x" "z"
```

17. Get all elements greater than 3.

```
y <- c(1, 5, 7, 2, 6)
y[y>3]
```

```
## [1] 5 7 6
```

18. Get all elements greater than 3 but less than 7.

```r
y <- c(1, 5, 7, 2, 6)

y[3<y & y < 7] #returns 5, 6
```

```
## [1] 5 6
```

19. Get all indices of elements greater than 3 but less than 7.

Use the function *which()* that returns a vector of indices.

```r
y <- c(1, 5, 7, 2, 6)
y[y>3]
```

```
## [1] 5 7 6
```

The `which` function is really helpful because it returns the index value

```r
y    <- c(1, 5, 7, 2, 6)
num5 <- which(y == 5)

y[num5]
```

```
## [1] 5
```

His example:

```r
which(y<3 & y<7) #returns index 2 and index 5 in vector y
```

```
## [1] 1 4
```

It's really worth nothing that the following operators return all true values:

- &
- |

Wheras the following only returns a single value

- &&
- ||

20. Get all sum of elements greater than 3 but less than 7.

```
y <- c(1, 5, 7, 2, 6)
sum(y[which(y>3 & y<7)]) #or sum(y[y>3 & y<7\])**
```

```
## [1] 11
```

```
# Returning only a single value
sum(y[which(y>3 && y<7)]) #or sum(y[y>3 & y<7\])**
```

```
## [1] 0
```

21. Add one to all numbers not divisible by 3.

If I wanted to add all numbers not divisible by 3 the following code will acheive that

```
y <- c(1, 5, 7, 2, 6)
y[which(y%%3!=0)] <- y[which(y%%3!=0)]+1
y
```

```
## [1] 2 6 8 3 6
```

`which(y%%3!=0)` returns vector of indices whose y values are not divisible by 3. After the operation y will be 2, 6, 8, 3, 6

21. Replace those numbers not divisible by 3 by NA (Not a number).

```
y <- c(1, 5, 7, 2, 6)
y[which(y%%3!=0)] <- NA # y will be NA NA NA NA 6
y
```

```
## [1] NA NA NA NA 6
```

22. Find the count of numbers in the vector.

```
# One Line
(y <- c(1, 5, 7, 2, 6)) %>% length()
```

```
## [1] 5
```

```
# Two Lines
y <- c(1, 5, 7, 2, 6)
y %>% length()
```

```
## [1] 5
```

Some other useful functions for working with vectors of data are:

Command	Description
`sd()`	standard deviation
`median()`	median
`max()`	maximum
`min()`	minimum
`range()`	maximum and minimum
`length()`	number of elements of a vector
`cummin()`	cumulative min (or max cummax())
`diff()`	differences between successive elements of a vector

Loops in R.

22. Print each number on a new line

Notice that `cat` does it in the background, it's a little quicker maybe in a loop :shrug:.

```
y <- c(1, 5, 7, 2, 6)

for (i in 1:length(y)){ cat(y[i],"\n")}
```

```
## 1
## 5
## 7
## 2
## 6
```

```
for (i in seq_len(length(y))) {

    cat(y[i], "\n")
  }
```

```
## 1
## 5
## 7
## 2
## 6
```

Generally however `1:length(...)` expressions should be avoided, instead use `seq_len`. An example of why is the backward sequence problem, e.g. a loop will run twice instead of zero times if you did `1:n` but `n=0`.

```
for (i in seq_len(length(y))) {

    cat(y[i], "\n")
  }
```

```
## 1
## 5
## 7
## 2
## 6
```

It is also possible to have the index variable take the form of the elements of a list:

```
for (i in 1:length(y)) {
 print(y[i])
}
```

```
## [1] 1
## [1] 5
## [1] 7
## [1] 2
## [1] 6
```

```
for(i in 1:length(y)){ cat(y[i],"\n")}
```

```
## 1
## 5
## 7
## 2
## 6
```

```
for(i in y){
  cat(y)
}
```

```
## 1 5 7 2 61 5 7 2 61 5 7 2 61 5 7 2 61 5 7 2 6
```

```
for(i in y) {
    cat(i, "\n")
}
```

```
## 1
## 5
## 7
## 2
## 6
```

While Loops

```
i <- 1

while(i <= length(y)) {
    cat(y[i], "\n")
    i <- i+1
}
```

```
## 1
## 5
## 7
## 2
## 6
```

mapping

```
library(purrr)
purrr::map(y, .f = ~ .x + 2) %>% as.vector()
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 4
##
```

```
## [[5]]
## [1] 8
```

There are generally two types of loops: * `for` loops are used when the number of repetitions is known + for every element in this list * `while` loops are used when the number of repetitions is not known + while this condition is false * There are other other types of loops in ***R*** + `purr::map()` maps the elements of a list to another list and transforms them accordingly, generally this is much faster than using a loop (in *Mathematica* it was about 10 fold faster generally). + `replicate` is a base function that runs a function continuously building a vector of the returned value, it runs faster than a loop and is a little like `table`. + using a static vector in a `for` loop is about 7 times faster (and there is no reason not to do it like in a `while` loop), using the `replicate` function might stop you from making the static/dynamic mistake but is otherwise just as fast. + `replicate` is a wrapper for `sapply` which is a wrapper for `lapply` so might be faster because it is a `.Internal` function written in ***C***.

Or

Qualitative Variables

Qualitative data, or categorical data is stored in R as factors. We can use the function factor() to coerce (convert) a vector to a factor

23. Given the character vector, find its factors

With factors it is necessary to be mindful to:

1. List levels where necessary
   - The `base::uniquie()` command is pretty helpful for this
2. Order Factors when necessary

Remember, when dealing with plots and the like it is often necessary to use factors in order to get correct grouping.

```
(grades <- c("HD", "F", "PA", "PA", "CR", "DI", "DI")) %>% class()
```

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

```
base::unique(grades)
```

```
## [1] "HD" "F" "PA" "CR" "DI"
```

```
grade_levels <- c("F", "PA", "CR", "DI", "HD")
grades <- factor(grades, levels = grade_levels, ordered = TRUE)
grades
```

```
## [1] HD F PA PA CR DI DI
## Levels: F < PA < CR < DI < HD
```

24. Given the character vector, find the structure of how it is stored

in r.

```
str(grades)
```

```
## Ord.factor w/ 5 levels "F"<"PA"<"CR"<..: 5 1 2 2 3 4 4
```

R Stores factors as numbers, it doesn't seem to matter if it's ordered or not:

```
factor(c("HD", "F", "PA", "PA", "CR", "DI", "DI")) %>% str()
```

```
## Factor w/ 5 levels "CR","DI","F",..: 4 3 5 5 1 2 2
```

26. Given the character vector, change the annotations for the levels.

```
table(grades) # %>% hist()
```

```
## grades
##  F PA CR DI HD
##  1  2  1  2  1
```

```
levels(grades) <- c("Fail", "Pass", "Credit", "Distinction", "High Distinction")
table(grades)
```

```
## grades
##             Fail             Pass           Credit      Distinction High Distinction
##                1                2                1                2                1
```

Note that `levels()` function can be used to change the annotations of the levels as well as to get the existing levels.

27. Forcing the levels to be in a specific order.

```
grades <- c("HD", "F", "PA", "PA", "CR", "DI", "DI")
grades.o.f <- factor(grades, levels= c("F", "PA", "CR", "DI", "HD"), ordered = TRUE)
levels(grades) <- c("Fail", "Pass", "Credit", "Distinction", "High Distinction")
grades.o.f
```

```
## [1] HD F PA PA CR DI DI
## Levels: F < PA < CR < DI < HD
```

[1] HD F PA PA CR DI DI

Levels: F $<$ PA $<$ CR $<$ DI $<$ HD

Notice that the levels are listed from lowest to highest, and are shown with the "$<$" indicating the order

28. Given the character vector find the frequency of each level.

$>$ grades <- c("HD", "F", "PA", "PA", "CR", "DI", "DI")

$>$ table(grades)

grades

CR DI F HD PA

1 2 1 1 2

There is 1 CR, 2 Dis, 1 F, 1 HD and 2 PAs.

Visualizing Qualitative Variables

We can use *barplot* to provide a graphical summary of a factor. However, we need to use **table()** with barplot(), since barplot() requires the values it will plot(its height argument) in a numeric form

Using the Sample Function

Refer to pages 14-16 in R book

[New Online Courses](https://newonlinecourses.science.psu.edu/stat484/sites/onlinecourses.science.psu.edu.stat484/files/filepicker/E
[1]

The function *sample()* will randomly choose values.

```
grades <- c("HD", "F", "PA", "PA", "CR", "DI", "DI")
```

29. Given the character vector what does the following code do?

```
# This will take a sample of the
sample(x = grades, size = 99, replace = TRUE, prob = (c(0.1, 0.1, 0.4, 0.1, 0.1,
    0.1, 0.1)) ) %>% table() # %>% hist()
```

[1]Refer to https://newonlinecourses.science.psu.edu/stat484/sites/onlinecourses.science.psu.edu.stat484/files/filepicker/EssentialR/index.pdf**%5D(https://newonlinecourses.science.psu.edu/stat484/sites/onlinecourses.science.psu.edu.stat484/files/filepicker/EssentialR/index.pdf

```
## .
## CR DI F HD PA
## 13 12 10 7 57
```

```
sample(grades)
```

```
## [1] "F" "DI" "CR" "HD" "PA" "DI" "PA"
```

It shuffles the grades and returns them. Hence each call to sample may return a different order of grades.

Working with numeric data

Getting summary of a data set using mean, median, sd, summary and quantile functions. A quantile, or percentile, tells you how much of your data lies below a certain value.

```
textData <- "10 23 34 17 29 40 56 78 90 67"
data    <- scan(text=textData) #parse string data
mean(data)
```

```
## [1] 44.4
```

```
var(data)
```

```
## [1] 734.4889
```

```
sd(data)
```

```
## [1] 27.10146
```

```
median(data)
```

```
## [1] 37
```

```
summary(data)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.00   24.50   37.00   44.40   64.25   90.00
```

```
quantile(data, c(0.25, 0.75))
```

```
##   25%   75%
## 24.50 64.25
```

25% of the data is likely to be below 24.5

75% of the data is likely to be below 64.25

50% of the data is likely to be between 24.5 and 64.25

```
quantile(data, c(0.18, 0.36, 0.54, 0.72, 0.9))
```

```
##   18%   36%   54%   72%   90%
## 20.72 30.20 39.16 61.28 79.20
```

Notice that the function quantile() can return any desired quantile.

The Data Frame (The R equivalent of the spreadsheet)

A data frame is a special type of list - it is a list of vectors that have the same length, and whose elements correspond to one another - i.e. the 4th element of each vector correspond. Think of it like a small table in a spreadsheet, with the columns corresponding to each vector, and the rows to each record.

```
names<- c("Chris", "Sue", "Ann", "Pat")
ages<- c(20, 30, 26, 40)
heights<- c(1, 3, 5, 8)
isStudent<- c(TRUE, FALSE, TRUE, TRUE)
cohort.df <- data.frame(names,ages,heights,isStudent)
cohort.df
```

```
##   names ages heights isStudent
## 1 Chris   20       1      TRUE
## 2   Sue   30       3     FALSE
## 3   Ann   26       5      TRUE
## 4   Pat   40       8      TRUE
```

30. Given the following data frame, find just the names of the people.

```
cohort.df$names
```

```
## [1] Chris Sue Ann   Pat
## Levels: Ann Chris Pat Sue
```

31. Given the following data frame, get all the details of the 1st and

3rd person.

```
cohort.df[c(1,3),]
```

```
##   names ages heights isStudent
## 1 Chris   20       1      TRUE
## 3   Ann   26       5      TRUE
```

32. Given the following data frame, get all the details of the first three persons.

```
cohort.df[1:3, ]
```

```
##   names ages heights isStudent
## 1 Chris   20       1      TRUE
## 2   Sue   30       3     FALSE
## 3   Ann   26       5      TRUE
```

33. Given the following data frame, get all the details of those

persons whose age >= 30.

```
cohort.df[cohort.df$ages >= 30,]
```

```
##   names ages heights isStudent
## 2   Sue   30       3     FALSE
## 4   Pat   40       8      TRUE
```

34. Given the following data frame, add another column representing the

city of each person

```
city <- vector(length = nrow(cohort.df))
city <- c("Sydney", "Melbourne", "Canberra", "Darwin")
cohort.df <- cbind(cohort.df, city)
```

34. Given the following data frame, add two more rows of data

corresponding to two persons. Assume your own data.

Base Packages

This is the way to do it, if you stuff around with vectors or lists you will invariably run into errors with factors, create a new data frame with the desired rows and then use `rbind` to attach that row. It doesn't matter really when adding columns, just use a vector, because a column will simply be a vector with a single data type anyway.

Speaking of which, be careful when using `df$name` notation, in `data frames` this will extract a vector, in `tibbles` this will extract a nX1 `tibble`.

```
# Use Lists so that the data can be entered row wise
  # Vectors are columns and cannot have different Data classes
row_1 <- data.frame("John", 23, 5, TRUE, "Darwin")
names(row_1) <- names(cohort.df)

row_2 <- data.frame("Tim", 28, 2, FALSE, "Perth")
names(row_2) <- names(cohort.df)

# Add the newdata to the Data Frame
rbind(cohort.df, row_1, row_2)
```

```
##   names ages heights isStudent   city
## 1 Chris   20       1      TRUE    Sydney
## 2   Sue   30       3     FALSE Melbourne
## 3   Ann   26       5      TRUE  Canberra
## 4   Pat   40       8      TRUE    Darwin
## 5  John   23       5      TRUE    Darwin
## 6   Tim   28       2     FALSE     Perth
```

TidyVerse

One of the advantages to using `dplyr::add_row()` is that ***R*** auto-completion (if you are using NCM2 and NCM-R in neovim) is that it will auto suggest the entry to add and the data type of the entry.

```
cohort.df <- cohort.df %>%
   dplyr::add_row(names = "Timmy",
          ages = 23,
          heights = 3,
          isStudent = TRUE,
          city = "Adelaide")
cohort.df
```

```
##   names ages heights isStudent   city
## 1 Chris  20       1      TRUE    Sydney
## 2   Sue  30       3     FALSE Melbourne
## 3   Ann  26       5      TRUE  Canberra
## 4   Pat  40       8      TRUE    Darwin
## 5 Timmy  23       3      TRUE  Adelaide
```

36. Assuming you have file named "BeanData.csv" in your working

directory, read the file.

```
# beans \<- read.table(\"BeansData.csv\",comm=\"\#\",header=TRUE,sep=\",\")
businessData <- read.csv(file =
    "https://www.stats.govt.nz/assets/Uploads/Business-price-indexes/Business-price-indexes-Decembe
    header = TRUE, sep = ",")
str(businessData)
```

```
## 'data.frame':  65748 obs. of 12 variables:
##  $ Series_reference: Factor w/ 780 levels "CEPQ.S2371","CEPQ.S2381",..: 1 1 1 1
    1 1 1 1 1 1 ...
##  $ Period          : num  1996 1997 1997 1997 1997 ...
##  $ Data_value      : num  899 884 925 932 929 940 956 958 970 1000 ...
##  $ Status          : Factor w/ 2 levels "FINAL","REVISED": 1 1 1 1 1 1 1 1 1 1 ...
##  $ UNITS           : Factor w/ 2 levels "Index","Percent": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Subject         : Factor w/ 4 levels "Capital Goods Price Index - CEP",..: 1 1
    1 1 1 1 1 1 1 1 ...
##  $ Group           : Factor w/ 18 levels "Energy Price Indexes - Base Period
    December 1996 quarter (=1000)",..: 15 15 15 15 15 15 15 15 15 15 ...
##  $ Series_title_1  : Factor w/ 471 levels "Accident and health insurance
    services",..: 186 186 186 186 186 186 186 186 186 186 ...
##  $ Series_title_2  : Factor w/ 3 levels "","Percentage change from previous
    period",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Series_title_3  : logi NA NA NA NA NA NA ...
##  $ Series_title_4  : logi NA NA NA NA NA NA ...
##  $ Series_title_5  : logi NA NA NA NA NA NA ...
```

```
summary(businessData)
```

```
##        Series_reference   Period      Data_value       Status        UNITS
## PPIQ.SQN800000 : 169  Min.   :1977  Min.   :  -5.8  FINAL :65732  Index :64845
## PPIQ.SQN900000 : 169  1st Qu.:2000  1st Qu.: 751.9  REVISED:  16  Percent: 903
## PPIQ.SQU900000 : 169  Median :2008  Median : 974.2
## PPIQ.SQN900000PC: 168  Mean   :2007  Mean   : 926.0
## PPIQ.SQU900000PC: 168  3rd Qu.:2014  3rd Qu.:1066.0
## PPIQ.SQN900000AC: 165  Max.   :2019  Max.   :3512.0
## (Other)        :64740                NA's   :400
##                        Subject
## Capital Goods Price Index - CEP: 7562
## Energy Statistics - NRG     : 1219
## Farm Inputs - FPI           :16027
## Producers Price Index - PPI :40940
##
##
##
##                                                            Group
## Farm expense price index - Expense categories - (Base Dec 2013 = 1000):16027
## Inputs (ANZSIC06) - NZSIOC level 3, Base: Dec. 2010 quarter (=1000) : 5621
## Inputs (ANZSIC06) - NZSIOC level 4, Base: Dec. 2010 quarter (=1000) : 5612
## Outputs (ANZSIC06) - NZSIOC level 4, Base: Dec. 2010 quarter (=1000) : 5356
## Outputs (ANZSIC06) - NZSIOC level 3, Base: Dec. 2010 quarter (=1000) : 5253
## Published output commodities, Base Dec 2009                    : 4558
## (Other)                                                        :23321
##                                              Series_title_1
## All Industries                                      : 1004
## Forestry and Logging                                :  618
## Mining                                              :  618
## Owner-Occupied Property Operation (National Accounts Only): 618
## Printing                                            :  618
## Wholesale Trade                                     :  618
## (Other)                                             :61654
##                                     Series_title_2 Series_title_3
    Series_title_4
##                                         :64845  Mode:logical  Mode:logical
## Percentage change from previous period      :  456  NA's:65748    NA's:65748
## Percentage change from same period previous year: 447
##
##
##
##
## Series_title_5
## Mode:logical
## NA's:65748
##
##
##
##
##
```

```
head(businessData)
```

```
##   Series_reference Period Data_value Status UNITS                  Subject
## 1      CEPQ.S2371 1996.12      899  FINAL Index Capital Goods Price Index - CEP
## 2      CEPQ.S2371 1997.03      884  FINAL Index Capital Goods Price Index - CEP
## 3      CEPQ.S2371 1997.06     925  FINAL Index Capital Goods Price Index - CEP
## 4      CEPQ.S2371 1997.09     932  FINAL Index Capital Goods Price Index - CEP
## 5      CEPQ.S2371 1997.12     929  FINAL Index Capital Goods Price Index - CEP
## 6      CEPQ.S2371 1998.03     940  FINAL Index Capital Goods Price Index - CEP
##                                                                      Group
## 1 Price Index by Item - Plant, Machinery and Equipment (Base: Sept 1999 = 1000)
## 2 Price Index by Item - Plant, Machinery and Equipment (Base: Sept 1999 = 1000)
## 3 Price Index by Item - Plant, Machinery and Equipment (Base: Sept 1999 = 1000)
## 4 Price Index by Item - Plant, Machinery and Equipment (Base: Sept 1999 = 1000)
## 5 Price Index by Item - Plant, Machinery and Equipment (Base: Sept 1999 = 1000)
## 6 Price Index by Item - Plant, Machinery and Equipment (Base: Sept 1999 = 1000)
##            Series_title_1 Series_title_2 Series_title_3 Series_title_4
    Series_title_5
## 1 Glass and glass products                      NA            NA            NA
## 2 Glass and glass products                      NA            NA            NA
## 3 Glass and glass products                      NA            NA            NA
## 4 Glass and glass products                      NA            NA            NA
## 5 Glass and glass products                      NA            NA            NA
## 6 Glass and glass products                      NA            NA            NA
```

37. Assuming you have initialized beans by importing from an Excel

file, find how many rows & columns of data exists.

```
dim(businessData)
```

```
## [1] 65748   12
```

38. Observe the first 6 rows of data in *beans*.

```
head(businessData
```

```
## Error: <text>:2:0: unexpected end of input
## 1: head(businessData
##    ^
```

39. Use the *apply* function to find the means of the last three

columns of data in beans.

First Observe when filtering data that the following are equivalent:

```
businessData[1:6, c("Period", "Data_value")]
```

```
##    Period Data_value
## 1 1996.12        899
## 2 1997.03        884
## 3 1997.06        925
## 4 1997.09        932
## 5 1997.12        929
## 6 1998.03        940
```

```
businessData[1:6 ,names(businessData) == c("Period", "Data_value")]
```

```
## data frame with 0 columns and 6 rows
```

In order to take the column mean values:

```
apply(businessData[1:6, c("Period", "Data_value")] , 2, mean)
```

```
##     Period Data_value
##  1997.0750  918.1667
```

We need to find the mean of all rows of column 6, 7 and 8 (MARGIN=2 refers to columns). So we use the *apply* function to repeatedly call the *mean* function. We use na.rm = TRUE to ignore NAs if any in the data while finding the mean.

40. Observe that the beans data has trt representing certain grouping.

If we are interested in find the mean of rt but grouped based on trt, then we can use tapply.

The `tapply` function is useful when `apply` needs to be used on data that needs to be split up first, it's like using `table` and apply together, for instance, consider the mileage of cars:

```
head(mtcars)
```

```
##                    mpg cyl disp hp drat   wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46 0 1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02 0 1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61 1 1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44 1 0    3    1
## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22 1 0    3    1
```

```
table(mtcars[,c("mpg", "cyl")]) %>% head()
```

```
##      cyl
## mpg   4 6 8
##   10.4 0 0 2
##   13.3 0 0 1
##   14.3 0 0 1
##   14.7 0 0 1
##   15   0 0 1
##   15.2 0 0 2
```

If the average mileage for each corresponding cylinder count was desired, it would be possible to do something like `table %>% as.matrix %>% apply`, however this has already been put into a wrapper with the `tapply` function:

```
tapply(mtcars$mpg, INDEX = mtcars$cyl, FUN = mean)
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

41. Using `tapply` for multiple

pot.size and P.lev (That is for: pot.size-4 and P.lev = L; pot.size-4 and P.lev = H; pot.size-8 and P.lev = L; pot.size-8 and P.lev = H; pot.size-12 and P.lev = L; pot.size-12 and P.lev = H;)

```
mtcars_round <- mtcars
mtcars_round$qsec <- signif(mtcars$qsec, 1)
head(mtcars_round)
```

```
##                    mpg cyl disp  hp drat    wt qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620   20  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875   20  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320   20  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215   20  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440   20  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460   20  1  0    3    1
```

```
table(mtcars_round[,c("mpg", "qsec")]) %>% head()
```

```
##      qsec
## mpg    10 20
##   10.4  0  2
##   13.3  0  1
##   14.3  0  1
##   14.7  0  1
##   15    1  0
##   15.2  0  2
```

```r
table(mtcars_round[,c("mpg", "cyl")]) %>% head()
```

```
##       cyl
## mpg    4 6 8
##   10.4 0 0 2
##   13.3 0 0 1
##   14.3 0 0 1
##   14.7 0 0 1
##   15   0 0 1
##   15.2 0 0 2
```

```r
table(mtcars_round[,c("mpg", "qsec", "cyl")])
```

```
## , , cyl = 4
## 
##       qsec
## mpg    10 20
##   10.4  0  0
##   13.3  0  0
##   14.3  0  0
##   14.7  0  0
##   15    0  0
##   15.2  0  0
##   15.5  0  0
##   15.8  0  0
##   16.4  0  0
##   17.3  0  0
##   17.8  0  0
##   18.1  0  0
##   18.7  0  0
##   19.2  0  0
##   19.7  0  0
##   21    0  0
##   21.4  0  1
##   21.5  0  1
##   22.8  0  2
##   24.4  0  1
##   26    0  1
##   27.3  0  1
##   30.4  0  2
##   32.4  0  1
##   33.9  0  1
## 
## , , cyl = 6
## 
##       qsec
## mpg    10 20
##   10.4  0  0
##   13.3  0  0
##   14.3  0  0
##   14.7  0  0
##   15    0  0
##   15.2  0  0
##   15.5  0  0
##   15.8  0  0
##   16.4  0  0
##   17.3  0  0
##   17.8  0  1
##   18.1  0  1
##   18.7  0  0
##   19.2  0  1
##   19.7  0  1
##   21    0  2
##   21.4  0  1
##   21.5  0  0
##   22.8  0  0
##   24.4  0  0
##   26    0  0
##   27.3  0  0
##   30.4  0  0
##   32.4  0  0
##   33.9  0  0
##
## , , cyl = 8
##
##       qsec
## mpg   10 20
##   10.4  0  2
##   13.3  0  1
##   14.3  0  1
##   14.7  0  1
##   15    1  0
##   15.2  0  2
##   15.5  0  1
##   15.8  1  0
##   16.4  0  1
##   17.3  0  1
##   17.8  0  0
##   18.1  0  0
##   18.7  0  1
##   19.2  0  1
##   19.7  0  0
##   21    0  0
##   21.4  0  0
##   21.5  0  0
##   22.8  0  0
##   24.4  0  0
##   26    0  0
##   27.3  0  0
##   30.4  0  0
##   32.4  0  0
##   33.9  0  0
```

```
tapply(mtcars$mpg, INDEX = list(signif(mtcars$qsec, 1), mtcars$cyl), mean)
```

```
##          4       6    8
## 10       NA       NA 15.40
## 20 26.66364 19.74286 15.05
```

42. The above result is in the form of a table. If we wanted a list instead, we can use the *paste* function.

I don't like this though, it's basically just flattening the matrix.

```
tapply(mtcars$mpg, INDEX = list(paste(signif(mtcars$qsec, 1), mtcars$cyl)), mean)
```

```
##     10 8     20 4     20 6     20 8
## 15.40000 26.66364 19.74286 15.05000
```

43. Write a function to return the square of a number:

You can specify a function where the last value is what will be returned:

```
sq_num <- function (x, force = FALSE) {

    if (is.numeric(x)) {
     x*x
    } else if (force) {
       x*x
    } else {
    print("Square Function is only defined for number")
    }

}

sq_num(3)
```

```
## [1] 9
```

```
sq_num(matrix(1:4, nrow = 2), force = TRUE)
```

```
##      [,1] [,2]
## [1,]    1    9
## [2,]    4   16
```

or you can deliberately declare it with `return()`, just be mindful that if you do need to use return it's possible that the function could be restructured to be more logical.

```
sq_num2 <- function (x) {
    xs <- x*x
    return(xs)
}
```

44. Custom Functions can be used with `tapply()`

Say we want to pool the sd between different engine classes but decide to see if they're close first:

```
my_mean <- function(x) {
    sum(x)/length(x-1) # x-1 because I'm using it with sample std. dev.
}
my_sd <- function(x) {
    xbar <- my_mean(x)
    sqrt(my_mean((x-xbar)^2))
}
tapply(mtcars$mpg, mtcars$cyl, my_sd)
```

```
##        4        6        8
## 4.299952 1.345742 2.466924
```

```
tapply(mtcars$mpg, mtcars$cyl, sd)
```

```
##        4        6        8
## 4.509828 1.453567 2.560048
```

My function's pretty close, I'm not sure where the difference comes from.

45. How to Perform Matrix Multiplication

```
(A <- matrix(c(1,2,3,4), ncol=2))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
(B <- matrix(c(10, 20, 30, 40), nrow = 2))
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

```
A*B
```

```
##      [,1] [,2]
## [1,]   10   90
## [2,]   40  160
```

```
A %*% B
```

```
##      [,1] [,2]
## [1,]   70  150
## [2,]  100  220
```

46. How to Diagonalise a Matrix

```
X<- c(1,2,3)
(XMat <- diag(X))
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    2    0
## [3,]    0    0    3
```

47. Normalize a Vector

How to normalize a vector. Normalizing a vector is obtaining another unit vector in the same direction.** **To normalize a vector, divide the vector by its magnitude.

```
vv<- c(1,5, 8, 10)
vv.squares <- vv^2
sum.of.vv.squares <- sum(vv.squares)
magnitude.vv<-sqrt(sum.of.vv.squares)
norm.vv <- vv/magnitude.vv
norm.vv
```

```
## [1] 0.07254763 0.36273813 0.58038100 0.72547625
```

```
sqrt(sum(norm.vv^2)) #verify this should be 1
```

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

48. Normalize a Matrix

How to normalize a matrix. Assume each column in a matrix is considered as a vector

```
(mx <- matrix(c(1,2,3,10,20,30,110,120,130),nrow=3, byrow = TRUE))
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   10   20   30
## [3,]  110  120  130
```

```
(norm.mx2 <- mx %*% diag(1/sqrt(colSums(mx ^2)))) %>% print(1)
```

```
##       [,1] [,2] [,3]
## [1,] 0.009 0.02 0.02
## [2,] 0.091 0.16 0.22
## [3,] 0.996 0.99 0.97
```

```
# Verify each column sums to 1
sqrt(colSums(norm.mx2^2)) #verify each column is a unit vector
```

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

References

New Online Courses PennState