1 Introduction

1.1 Types of Analysis

- **Descriptive Analytics**: What happened? What is happening?
- **Predictive Analytics**: What will happen?
- **Prescriptive Analytics**: What to do?

2 R Basics

2.1 Working Environment

- **Workspace**: contains all variables and functions (collectively known as **objects**) as well as any packages loaded
- **Working directory**: Default directory where R will look for files loaded/stored
- **Project**: Data, R scripts, analytical results, and figures about a particular problem are normally organised and stored under one project

2.1.1 Relavent Commands

- `getcwd()`: get working directory
- `setwd(<dir>)`: sets the working directory to <dir>
- `ls()`: lists all objects (variables and functions) defined in the current work space
  - We can use `rm(<obj>)` to remove
- `dir()`: lists all files and subfolders in cwd
  - `list.files()` does the same as `dir()`
- `file.exists(<file>)`: checks if a certain file exist
  - Useful when we want to check if dir exists before creating it
  - `ifelse(!dir.exists("a"),dir.create("a"),"a exists")`

2.2 Atomic Data Types

R has 5 basic (atomic) data types

- **Logical**
  - TRUE or FALSE, T or F
  - In terminal output, will always be TRUE/FALSE
  - Return value of logical operators: <, >, |, &, !
- **Numeric**
  - Decimal values, e.g. 1.23
  - By default, if we assign integer to variable, the class will be numeric, e.g. `k<-2`, *k* will be numeric
  - We cannot cast strings to integer or numeric, will have warning of `NAs introduced by coercion`
- **Integer**
  - To convert numeric to integer, have to use the `as.integer(<num>)` function
  - Note that the `as.integer(<num>)` will always round down, i.e. 2.34 → 2 and 2.56 → 2
- **Character**
  - Basically string data type in R
  - Can use `nchar(<string>)` to find number of characters
  - To find index of matches within a string `regexpr(<pattern>, <string>)`, will return -1 if no results found e.g.

```
regexpr("ex", "longtext")
## [1] 6
## attr(,"match.length")
## [1] 2
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

  - Can use `gregexpr(<pattern>, <string>)` to find positions of every match
  - Can use `grep(<pattern>, <vector>)` to find the positions of a regular expression in a vector of text strings
  - Use `substr(<string>, <start>, <end>)` to get a slice of from start to end (inclusive). Strings are 1-indexed
  - Use `sub(<pattern>, <replacement>, <string>)` to replace the first match of a string with a new string and `gsub()` to replace all matches

2.3 Data Structures

2.3.1 Vectors

- ordered array of elements of the same data type
- Created using `arr<-c(1,2,3)`
- Data coercion to most flexible data type if trying to store multiple data types
  - In order of **boolean > int > numeric > character**
- Can check the type of vector using `typeof()`
- Can name vectors using:

```
a<-c(1,3,4)
furniture <- c("desks", "tables", "chairs")
names(a) <- furniture

a<-c("desks" = 1, "tables" = 3, "chairs" = 4)

a<-c(desks = 1, tables = 3, chairs = 4)
```

2.3.2 Vector Arithmetic

- `arr + 100` will add 100 to each element in the vector
- Can also do the standard + - * / ^ operations
- Math operations will not work on strings
- Can also do operations on 2 vectors
  - `c(1,2,3,4)+c(1,2,3)->c(2,4,6,5)` + warning (**recycling**)
- Possible to use `mean, prod, sum` methods too

2.3.3 Vector Subsetting

- As in python, use [] to access elements in vector
- Can access using their indexes or their name
- Access multiple elements by doing `materials[c(4,3)]`
- Possible to use `moreMetals[-6]` which selects all other indexes besides 6th element
- Can also index using `arr[c(TRUE, TRUE, FLALSE)]`, if inner boolean array < size of arr, recycling will happen, if size inner array > size of arr, NA will be returned

2.3.4 Matrices

- Matrix are arranged **by default by columns**, to change it to by row, use `matrix(3:8,ncol = 3,nrow = 2,byrow = TRUE)`
- Matrices are fundamentally vectors
- Can only contain homogeneous data types
- To create a matrix:

```
matrix(3:8,ncol=3,nrow=2)

matrix(3:8,nrow=2)

m2<-3:8
dim(m2)<-c(2,3)

##      [,1] [,2]
## [1,]    3    6
## [2,]    4    7
## [3,]    5    8
```

- If nrow × ncol > range of number supplied, recycling will happen

```
matrix(3:5,ncol=3,nrow=2)

##      [,1] [,2] [,3]
## [1,]    3    5    4
## [2,]    4    3    5
```

- If length of input arr not a multiple of nrow × ncol, warning will be thrown
- Can name columns and rows using

```
rownames(m3)<-c("Row1","Row2")
colnames(m3)<-c("Col.1","Col.2","Col.3")
```

- Matrix access is by [row, col]

2.3.5 Factors

- Factors are special variables used to store **categorical variables**
- Advantages of using factors:
  - Factor variables are stored as a vector of integer values, thus it is a more efficient use of memory
  - statistical models will automatically handle factor variables properly
  - useful in graphics
- To create factors:

```
a<-c(0,1,0,0,1)
a.f<-factor(a,labels = c("Male","Female"))

a<-c("One","Two","Three","One","Three")
a.f<-factor(a)
```

- Levels in factors are **ordered in lexicographical order**, have to manually assign them to avoid this

2.3.6 List

- Allows for multiple types to be stored in the same array
- Created using:

```
list(Name="Mike",Age=43,Children=c("Tom","Lily"))
```

- Can access elements using the standard [] operator or using the $ symbol, e.g. `lst$Age`
- Use the `str` command to display the internal structure of a list

```
str(Mike)

## List of 4
##  $ Name    : chr "Mike"
##  $ Salary  : num 10000
##  $ Age     : num 43
##  $ Children: chr [1:3] "Tom" "Lily" "Alice"
```

2.3.7 DataFrames

- Basically a matrix that can contain heterogeneous data types
- Is just a list of vectors, can use `length` to get the number of rows
- Created using the `data.frame()` method

```
name <- c("Anne", "Pete", "Frank", "Julia", "Cath")
age <- c(28, 30, 21 ,39, 35)
child <- c(FALSE, TRUE, TRUE, FALSE, TRUE)

df <- data.frame(name, age, child)
```

- Can also name the cols using `names(<df>)` function
- Will automatically convert character data type into factors
  - If we want to store them as characters, use `data.frame(name, age, child, stringsAsFactors=FALSE)`
- Subsetting is the same as in matrices, the following results are returned as a **Dataframe**

```
df[3,2] ## Select element in row 3, col 2
df[3,"age"] ## Select row 3, with col name of "age"
df[3,] ## select entire row 3
df[,"age"] ## select entire col with name "age"
df[c(3,5), c("age", "child")]
```

- If we want to return them as a **list**

```
df$age
df[["age"]]
df[[2]]
```

- We can extend dataframes using `cbind()` or `rbind()`
- Sorting dataframes
  - `sort(df$age)` return ascending order of "age" column
  - `order(df$age)` return order of the current indexes if sorted
  - `max(df$age)` returns max **value** of "age" column
  - `which.max(df$age)` return **index** of element with max value in "age" column
  - `rank(df$age)` return current ranking of each element
- Indexing dataframe
  - `df[df$age > 30 & df$child == FALSE,]`
  - `which(df$name == "Cath")` returns index where name == "Cath"
  - `match(c("Anne", "Julia", "Cath"), df$name)` return indexes that match those names `c("Anne", "Julia", "Cath", "Bob") %in% df$name`

3 Data Wrangling

3.1 Built-in

- `read.csv(<file>)` to read CSV files
- `read.csv2(<file>)` to read semicolon seperated file
- `read.delim(<file>)` to specify the delimeter
- `summary(<matrix>)` returns min, 1st quartile, median, mean, 3rd quartile, max and number of NAs for each col

3.2 Dplyr

- `mutate(df, bmi = weight/height^2*10000)` add a new col bmi
- `filter(df, bmi > 18.5 & bmi < 24.9)` filters out rows that match the condition
- `select(df, name, height)` select name and heigh col of df
- `intersect(<arr>, <arr>)` returns common element between 2 vectors/dataframe
- `union(<arr>, <arr>)` returns the union of the 2 vectors/-dataframes, taking into account of duplicates
- `setdiff(<arr>, <arr>)`, usage:

```
setdiff(1:10, 6:15) ## [1] 1 2 3 4 5
setdiff(6:15, 1:10) ## [1] 11 12 13 14 15
```

- `setequal(<arr>, <arr>)` returns whether 2 sets/dataframes are same, regardless of order

3.3 readr

Function	Format	Typical Suffix
read_table	whitespace separated values	txt
read_csv	comma separated values	csv
read_csv2	semicolon separated values	csv
read_tsv	tab delimited separated values	tsv
read_delim	general text file format, must specify delimiter	txt

3.4 readxl

Function	Format	Typical Suffix
read_excel	auto detects the format	xls, xlsx
read_xls	original format	xls
read_xlsx	new excel format	xlsx

3.5 jsonlite

```
url <- "https://api.data.gov.sg/v1/carpark-availability"
data <- fromJSON(url)
a <- as.data.frame(data$items$carpark_data)
```

3.6 XML

```
data <- xmlParse("books.xml")
root <- xmlRoot(data) # get root node
nodes <- xmlChildren(root) # get child nodes of root
a <- nodes[[2]] # get 2nd child of root
books <- getNodeSet(data, "/catalog/book[@type='HardCove
xmlToList(date)
xmlToDataFram(books)
```

3.7 tidyr

Wide data format

Time	A	B	C
0	1.1	4.2	5.6
1	1.0	4.5	5.8

Tidy data format

Time	Sample	Value	id
0	A	1.1	1
1	A	1.0	1
0	B	4.2	1
1	B	4.5	1
0	C	5.6	1
1	C	5.8	1

- Main use is to reshape data, from wide to tidy
- `wide_data %>% gather(year, fertility, '1960':'2015')`
  - first argument will be the name of column for the gathered variables
  - second argument is for the values in the column cells
  - third argument in the function to specify the specific columns to gather
- Can change from tidy to wide using `spread()` command

3.8 Joins

- `left_join(<table>, <table>, by=c('col'='col'))` join right into left with matching entries, keep everything in left
- `right_join(<table>, <table>, by=c('col'='col'))` join left into right with matching entries, keep everything in right
- `inner_join(<table>, <table>, by=c('col'='col'))` basically intersection of 2 tables
- `full_join(<table>, <table>, by=c('col'='col'))` basically a union of 2 tables
- `semi_join(<table>, <table>, by=c('col'='col'))` allows us to keep the part of the first table for which we have information in the second table, but doesnt add the columns of the second
- `anti_join(<table>, <table>, by=c('col'='col'))` allows us to keep the part of the first table for which we have NO information in the second table, but doesnt add the columns of the second

4 Programming Structure and Functions

1. Conditionals

```
if(boolean condition){
    expressions
} else{
    alternative expressions
}

ifelse(boolean condition,expressions,alt expressions)
```

2. `any(<vector>)`

   returns TRUE if any of logicals are true
3. `all(<vector>)`

   returns TRUE if all of logicals are true
4. Functions

```
my_function <- function(x, y, z=1){
    operations that operate on x, y, z
    value of final line is returned
}
```

5. For loops

```
for (i in range of values){
    operations that use i
}
```

6. `apply(x, MARGIN, FUNC, <args>)`

   applies FUNC over each element of a matrix/dataframe. Margin=1 will apply row wise, Margin=2 will apply col wise