R: An Introduction

The History of R

- R actually evolved from the **S** language, which was first developed by Rick Becker, John Chambers and Allan Wilks.

The History of R

- R actually evolved from the **S** language, which was first developed by Rick Becker, John Chambers and Allan Wilks.
- S is made up of three components.

The History of R

- R actually evolved from the **S** language, which was first developed by Rick Becker, John Chambers and Allan Wilks.
- S is made up of three components.
  1. S is a powerful tool for statistical modelling. It enables you to specify and fit statistical models to your data, assess the goodness of fit and display the estimates, standard errors and predicted values derived from the model. It provides the means to define and manipulate data. The user is left with maximum control over the model-fitting process.

The History of R

- R actually evolved from the **S** language, which was first developed by Rick Becker, John Chambers and Allan Wilks.
- S is made up of three components.
  1. S is a powerful tool for statistical modelling. It enables you to specify and fit statistical models to your data, assess the goodness of fit and display the estimates, standard errors and predicted values derived from the model. It provides the means to define and manipulate data. The user is left with maximum control over the model-fitting process.
  2. S can be used for data exploration (tabulating, sorting data, drawing plots to look for trends, etc.

The History of R

- R actually evolved from the **S** language, which was first developed by Rick Becker, John Chambers and Allan Wilks.
- S is made up of three components.
  1. S is a powerful tool for statistical modelling. It enables you to specify and fit statistical models to your data, assess the goodness of fit and display the estimates, standard errors and predicted values derived from the model. It provides the means to define and manipulate data. The user is left with maximum control over the model-fitting process.
  2. S can be used for data exploration (tabulating, sorting data, drawing plots to look for trends, etc.
  3. It can be used as a sophisticated calculator to evaluate complex arithmetic expressions, and a very flexible and general object-oriented programming language to perform more extensive data manipulation.

The History of R

- R actually evolved from the **S** language, which was first developed by Rick Becker, John Chambers and Allan Wilks.
- S is made up of three components.
  1. S is a powerful tool for statistical modelling. It enables you to specify and fit statistical models to your data, assess the goodness of fit and display the estimates, standard errors and predicted values derived from the model. It provides the means to define and manipulate data. The user is left with maximum control over the model-fitting process.
  2. S can be used for data exploration (tabulating, sorting data, drawing plots to look for trends, etc.
  3. It can be used as a sophisticated calculator to evaluate complex arithmetic expressions, and a very flexible and general object-oriented programming language to perform more extensive data manipulation.
- Later on, S evolved into S-PLUS, however, S-PLUS was very expensive.

The History of R

- R actually evolved from the **S** language, which was first developed by Rick Becker, John Chambers and Allan Wilks.
- S is made up of three components.
  1. S is a powerful tool for statistical modelling. It enables you to specify and fit statistical models to your data, assess the goodness of fit and display the estimates, standard errors and predicted values derived from the model. It provides the means to define and manipulate data. The user is left with maximum control over the model-fitting process.
  2. S can be used for data exploration (tabulating, sorting data, drawing plots to look for trends, etc.
  3. It can be used as a sophisticated calculator to evaluate complex arithmetic expressions, and a very flexible and general object-oriented programming language to perform more extensive data manipulation.
- Later on, S evolved into S-PLUS, however, S-PLUS was very expensive.
- **R**oss Ihaka and **R**obert Gentleman from the University of Auckland, decided to write a stripped-down version of S, which was R. Five years later, version 1.0.0 of R was released on 29 Feb 2000.

What is R?

It is an integrated suite of software facilitates for data manipulation, calculation and graphical display.
Among other things it has

- An effective data handling and storage facility.

What is R?

It is an integrated suite of software facilitates for data manipulation, calculation and graphical display.
Among other things it has

- An effective data handling and storage facility.
- A suite of operators for calculations on array, in particular, matrices.

What is R?

It is an integrated suite of software facilitates for data manipulation, calculation and graphical display.
Among other things it has

- An effective data handling and storage facility.
- A suite of operators for calculations on array, in particular, matrices.
- A large, coherent, integrated collection of intermediate tools for data analysis.

What is R?

It is an integrated suite of software facilitates for data manipulation, calculation and graphical display.
Among other things it has

- An effective data handling and storage facility.
- A suite of operators for calculations on array, in particular, matrices.
- A large, coherent, integrated collection of intermediate tools for data analysis.
- Graphical facilities for data analysis.

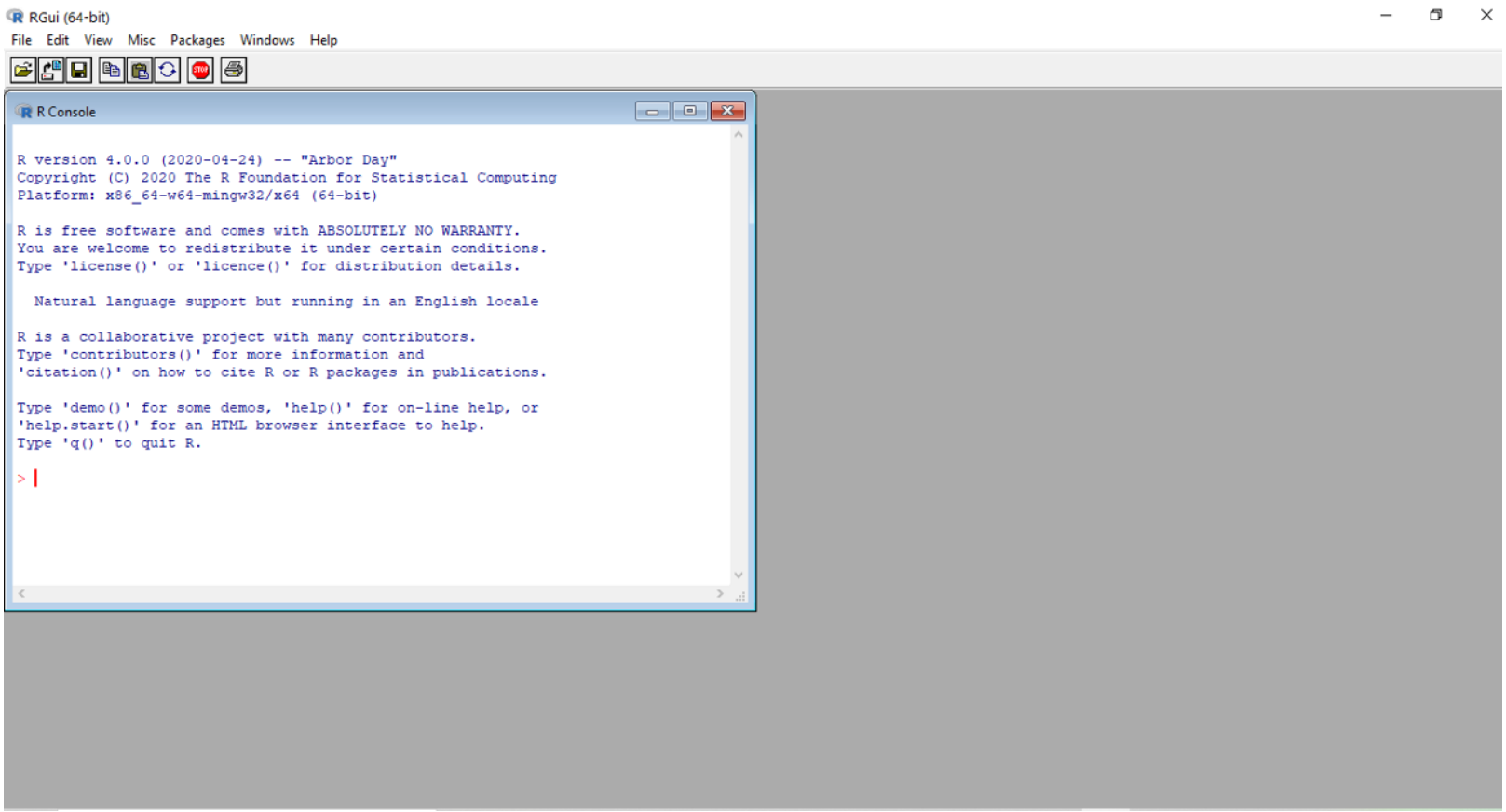
What is R?

It is an integrated suite of software facilitates for data manipulation, calculation and graphical display.
Among other things it has

- An effective data handling and storage facility.
- A suite of operators for calculations on array, in particular, matrices.
- A large, coherent, integrated collection of intermediate tools for data analysis.
- Graphical facilities for data analysis.
- A well developed, simple and effective programming language.

How To Start R?

- Downloading R from `https://www.r-project.org/`
- Double click the R's icon in the desktop to activate R (at least version 4.0.0).
- After R is started, the R console is open in the RGui window.



How To Handle Data in R

Four most frequently used types of data objects:

- Vector: set of elements of the same mode (logical; numeric; character).

How To Handle Data in R

Four most frequently used types of data objects:

- Vector: set of elements of the same mode (logical; numeric; character).

- Matrix: set of elements appearing in rows and columns, where the elements are of the same mode.

How To Handle Data in R

Four most frequently used types of data objects:

- Vector: set of elements of the same mode (logical; numeric; character).

- Matrix: set of elements appearing in rows and columns, where the elements are of the same mode.

- Dataframe:
  -Similar to the Matrix object but columns can have different modes.

  -The rows contain different observations from your study or measurements from your experiment;

  -The columns contain the values of different variables which may be of different modes.

How To Handle Data in R

Four most frequently used types of data objects:

- Vector: set of elements of the same mode (logical; numeric; character).
- Matrix: set of elements appearing in rows and columns, where the elements are of the same mode.
- Dataframe:
  -Similar to the Matrix object but columns can have different modes.

  -The rows contain different observations from your study or measurements from your experiment;

  -The columns contain the values of different variables which may be of different modes.
- List: generalization of a vector – represents a collection of data objects.

Creating a Vector in R: "c" function

- To create a vector, the simplest way is using the concatenation "c" function.

```
> #creating a vector of numbers:
> number<-c(2,4,6,8,10)
> number
[1]  2  4  6  8 10
> # creating a vector of strings/characters:
> string<-c("weight", "height", "gender")
> string
[1] "weight" "height" "gender"
> #creating a Boolean vector (T/F):
> logic<- c(T, T, F, F, T)
> logic
[1]  TRUE  TRUE FALSE FALSE  TRUE
```

Appending item(s) to the existing vector:

Creating a Vector in R: "c" function

- To create a vector, the simplest way is using the concatenation "c" function.

```
> #creating a vector of numbers:
> number<-c(2,4,6,8,10)
> number
[1]  2  4  6  8 10
> # creating a vector of strings/characters:
> string<-c("weight", "height", "gender")
> string
[1] "weight" "height" "gender"
> #creating a Boolean vector (T/F):
> logic<- c(T, T, F, F, T)
> logic
[1]  TRUE  TRUE FALSE FALSE  TRUE
```

Appending item(s) to the existing vector:

- What is c(number,12,14)? A vector of numbers.

Creating a Vector in R: "c" function

- To create a vector, the simplest way is using the concatenation "c" function.

```
> #creating a vector of numbers:
> number<-c(2,4,6,8,10)
> number
[1]  2  4  6  8 10
> # creating a vector of strings/characters:
> string<-c("weight", "height", "gender")
> string
[1] "weight" "height" "gender"
> #creating a Boolean vector (T/F):
> logic<- c(T, T, F, F, T)
> logic
[1]  TRUE  TRUE FALSE FALSE  TRUE
```

Appending item(s) to the existing vector:

- What is c(number,12,14)? A vector of numbers.
- What is c(string, 12,14)? A vector of strings where "12" and "14" are treated as strings.

Creating a Vector in R: "numeric" function

The "numeric" function creates a vector with all its elements being 0.

```
> number.2<-numeric(3)
> number.2
[1] 0 0 0
> c(number, number.2)
[1]  2  4  6  8 10  0  0  0
> c(string, number.2)
[1] "weight" "height" "gender" "0"      "0"      "0"
```

Creating a Vector in R: "rep" function

The "rep" function replicates elements of vectors.
rep(a,b): replicate the item *a* by *b* times.

```
> #rep(a,b): replicate the item a by b times.
> number.3<-rep(2,3)
> number.3
[1] 2 2 2
> number.3<-rep(c(1,2),3)
> number.3
[1] 1 2 1 2 1 2
> rep(c(6,3),c(2,4))#6 and 3 are replicated 2, 4 times, respectively
[1] 6 6 3 3 3 3
> rep(string,2)
[1] "weight" "height" "gender" "weight" "height" "gender"
```

What is the length of the vector: rep(c(6,3,2),c(2,1,2))?

Creating a Vector in R: "seq" function

seq(from = a, to = b, by = c): from the number *a* to number *b*, create a sequence of numbers evenly spread by a distance of *c*.

```
> seq(from=2, to=10, by=2)
[1]  2  4  6  8 10
> seq(from=2, to=10, length = 5)
[1]  2  4  6  8 10
> 1:5
[1] 1 2 3 4 5
> 1:5*2
[1]  2  4  6  8 10
> seq(2,10,2)
[1]  2  4  6  8 10
> seq(10)
 [1]  1  2  3  4  5  6  7  8  9 10
> c(10,8,4, rep(2,3), rep(1:2,2), rep(c(5,7),2:3), seq(6,10,2))
 [1] 10  8  4  2  2  2  1  2  1  2  5  5  7  7  7  6  8 10
```

Creating a Matrix: "dim" function

```
> v <- c(1:6)
> dim(v) <- c(2,3) # 2 rows, 3 columns
> v
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- The dim(v) command will take the values of the vector v to fill the matrix by column.

```
> dim(v)=NULL
> v
[1] 1 2 3 4 5 6
```

Creating a Matrix: "dim" function

```
> v <- c(1:6)
> dim(v) <- c(2,3) # 2 rows, 3 columns
> v
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- The dim(v) command will take the values of the vector v to fill the matrix by column.

- To convert back to a vector, we simply use the dim function again.

```
> dim(v)=NULL
> v
[1] 1 2 3 4 5 6
```

Creating a Matrix: "matrix" function

- *matrix(v,r,c)*: take the values from vector *v* to create a matrix with *r* rows and *c* columns.

```
> v <- c(1:6)
> m <- matrix(v, nrow=2, ncol=3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> # to fill the matrix by rows:
> m <- matrix(v, nrow=2, ncol=3, byrow=T)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Creating a Matrix: "matrix" function

- *matrix(v,r,c)*: take the values from vector *v* to create a matrix with *r* rows and *c* columns.
- By default, matrix is filled by column.

```
> v <- c(1:6)
> m <- matrix(v, nrow=2, ncol=3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> # to fill the matrix by rows:
> m <- matrix(v, nrow=2, ncol=3, byrow=T)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Creating a Matrix: "rbind" and "cbind" functions

- To bind a row (or many rows) onto a matrix, the command *rbind* can be used.

```
> a <- c(1,2,3,4)
> b <- c(5,6,7,8)
> ab_row <- rbind(a,b)
> ab_row
  [,1] [,2] [,3] [,4]
a    1    2    3    4
b    5    6    7    8
```

Creating a Matrix: "rbind" and "cbind" functions

- To bind a row (or many rows) onto a matrix, the command *rbind* can be used.

```
> a <- c(1,2,3,4)
> b <- c(5,6,7,8)
> ab_row <- rbind(a,b)
> ab_row
```

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

- To bind a column (or many columns) onto a matrix, the command *cbind* can be used.

```
> ab_col <- cbind(ab_row, c(9,10))
> ab_col
```

```
  [,1] [,2] [,3] [,4] [,5]
a    1    2    3    4    9
b    5    6    7    8   10
```

Form of a Dataframe in R (1)

- R handles data in objects known as dataframes.

Example: an experiment with three treatments (control, pre-heated and pre-chilled), and four measurements per treatment. A dataframe is created based on the given measurements.

This is a wrong dataframe.

Form of a Dataframe in R (1)

- R handles data in objects known as dataframes.
- A dataframe is an object with rows and columns:

Example: an experiment with three treatments (control, pre-heated and pre-chilled), and four measurements per treatment. A dataframe is created based on the given measurements.

This is a wrong dataframe.

Form of a Dataframe in R (1)

- R handles data in objects known as dataframes.
- A dataframe is an object with rows and columns:
  - The rows contain different **observations or measurements**;

Example: an experiment with three treatments (control, pre-heated and pre-chilled), and four measurements per treatment. A dataframe is created based on the given measurements.

This is a wrong dataframe.

Form of a Dataframe in R (1)

- R handles data in objects known as dataframes.

- A dataframe is an object with rows and columns:
  - The rows contain different **observations or measurements**;
  - The columns contain the values of different **variables**.

Example: an experiment with three treatments (control, pre-heated and pre-chilled), and four measurements per treatment. A dataframe is created based on the given measurements.

This is a wrong dataframe.

Form of a Dataframe in R (1)

- R handles data in objects known as dataframes.
- A dataframe is an object with rows and columns:
  - The rows contain different **observations or measurements**;
  - The columns contain the values of different **variables**.
- **All the values of the same variable must go in the same column**.

Example: an experiment with three treatments (control, pre-heated and pre-chilled), and four measurements per treatment. A dataframe is created based on the given measurements.

This is a wrong dataframe.

Form of a Dataframe in R (2)

The correct dataframe should be

Response	Treatment
6.1	Control
5.9	Control
5.8	Control
5.4	Control
6.3	Pre-heated
6.2	Pre-heated
5.8	Pre-heated
6.3	Pre-heated
7.1	Pre-chilled
8.2	Pre-chilled
7.3	Pre-chilled
6.9	Pre-chilled

- This has 2 variables: measurements as the response variable and another variable (called "treatment") for three levels of experimental factor.

Creating Dataframes in R: "data.frame" command (1)

data.frame converts a matrix or a collection of vectors into a dataframe.

- Example

```
> v <- c(1:6)*2
> v
[1]  2  4  6  8 10 12
> m <- matrix(v, 2, 3) #matrix of 2 rows, 3 columns
> df1 <- data.frame(m)
> df1
  X1 X2 X3
1  2  6 10
2  4  8 12
> names(df1)
[1] "X1" "X2" "X3"
```

Creating Dataframes in R: "data.frame" command (2)

- Another example

```
> a<- c(11,12)
> b <- c(13,14)
> df2 <- data.frame(a,b)
> df2
   a  b
1 11 13
2 12 14
> names(df2)
[1] "a" "b"
```

In this example, *a* and *b* (the vectors' name) are used for the column names in the dataframe instead of the default *X1* and *X2*.

Specifying Column/Row names

- The columns are automatically labeled.

```
> names(df1)
[1] "X1" "X2" "X3"
```

Specifying Column/Row names

- The columns are automatically labeled.

```
> names(df1)
[1] "X1" "X2" "X3"
```

- Column names can be re-named.

```
> names(df1)=c("Column 1","Column 2","Column 3")
> names(df1)
[1] "Column 1" "Column 2" "Column 3"
```

Specifying Column/Row names

- The columns are automatically labeled.

```
> names(df1)
[1] "X1" "X2" "X3"
```

- Column names can be re-named.

```
> names(df1)=c("Column 1","Column 2","Column 3")
> names(df1)
[1] "Column 1" "Column 2" "Column 3"
```

- Row names are automatically assigned and are labeled as "1", "2", and so on:

```
> row.names(df1)
[1] "1" "2"
```

Specifying Column/Row names

- The columns are automatically labeled.

```
> names(df1)
[1] "X1" "X2" "X3"
```

- Column names can be re-named.

```
> names(df1)=c("Column 1","Column 2","Column 3")
> names(df1)
[1] "Column 1" "Column 2" "Column 3"
```

- Row names are automatically assigned and are labeled as "1", "2", and so on:

```
> row.names(df1)
[1] "1" "2"
```

- Row names can be re-named if desired:

```
> row.names(df1)=c("Row1","Row2")
> row.names(df1)
[1] "Row1" "Row2"
```

Reading Data Files: A Quick Overview

There are several ways of reading/importing data files into R:

- *scan(...)* offers a low-level reading facility: read data into a vector or list from the console or file.

Reading Data Files: A Quick Overview

There are several ways of reading/importing data files into R:

- *scan(...)* offers a low-level reading facility: read data into a vector or list from the console or file.

- *read.table(...)* can be used to read dataframes from free format text files (.txt files).

Reading Data Files: A Quick Overview

There are several ways of reading/importing data files into R:

- *scan(...)* offers a low-level reading facility: read data into a vector or list from the console or file.
- *read.table(...)* can be used to read dataframes from free format text files (.txt files).
- *read.csv(...)* can be used to read dataframes from files using comma to separate values (.csv files).

Reading Data Files: A Quick Overview

There are several ways of reading/importing data files into R:

- *scan(...)* offers a low-level reading facility: read data into a vector or list from the console or file.
- *read.table(...)* can be used to read dataframes from free format text files (.txt files).
- *read.csv(...)* can be used to read dataframes from files using comma to separate values (.csv files).
- *read.fwf(...)* can be used to read files that have a fixed width format.

Reading Data Files: A Quick Overview

There are several ways of reading/importing data files into R:

- *scan(...)* offers a low-level reading facility: read data into a vector or list from the console or file.
- *read.table(...)* can be used to read dataframes from free format text files (.txt files).
- *read.csv(...)* can be used to read dataframes from files using comma to separate values (.csv files).
- *read.fwf(...)* can be used to read files that have a fixed width format.
- When reading from Excel files, a simple method is to save each worksheet separately as a csv file and use *read.csv(....)* on each saved csv file.

Importing a Dataset to R: "scan" function

scan() function read data into a vector or list from the console or file.

```
> v<-scan()
1: 1 2 3 4 5
6: 4 3 2 1
10: 1 1 1 1
14:
Read 13 items
> v
 [1] 1 2 3 4 5 4 3 2 1 1 1 1 1
>
```

Import a Free Format Data File (1)

- The first line contains the names of variables, then we use: *header* = *TRUE*. Without *header* = *TRUE*:

```
> data1<-read.table("C:/Data/crab.txt")
> data1
```

	V1	V2	V3	V4	V5
1	color	spine	width	satell	weight
2	3	3	28.3	8	3.050
3	4	3	22.5	0	1.550
4	2	1	26.0	9	2.300
5	4	3	24.8	0	2.100
6	4	3	26.0	4	2.600
7	3	3	23.8	0	2.100
8	2	1	26.5	0	2.350
9	4	2	24.7	0	1.900
10	3	1	23.7	0	1.950
11	4	3	25.6	0	2.150
12	4	3	24.3	0	2.150
13	3	3	25.8	0	2.650
14	3	3	28.2	11	3.050
15	5	2	21.0	0	1.850

Import a Free Format Data File (2)

- With *header* = *TRUE*.

```
> data1<-read.table("C:/Data/crab.txt", header = TRUE)
> data1
    color spine width satell weight
1       3     3  28.3      8  3.050
2       4     3  22.5      0  1.550
3       2     1  26.0      9  2.300
4       4     3  24.8      0  2.100
5       4     3  26.0      4  2.600
6       3     3  23.8      0  2.100
7       2     1  26.5      0  2.350
8       4     2  24.7      0  1.900
9       3     1  23.7      0  1.950
10      4     3  25.6      0  2.150
11      4     3  24.3      0  2.150
12      3     3  25.8      0  2.650
13      3     3  28.2     11  3.050
14      5     2  21.0      0  1.850
15      3     1  26.0     14  2.300
16      2     1  27.1      8  2.950
```

Import a Free Format Data File (2)

- If the first line of the data file does not contain the names of the variables, we can create a vector to store the variable names:

```
> varnames <- c("Subject", "Gender", "CA1", "CA2", "HW")
> data2<-read.table("C:/Data/ex_1.txt", header = FALSE,
+                   col.names = varnames)
> data2
```

```
  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
2       7      M  85  89  A
3       4      F  90  86  B
4      20      M  82  85  B
5      25      F  94  94  A
6      14      F  88  84  C
```

Missing values are denoted by NA.

Importing a Comma Separated Data

- The most convenient way to read in comma-separated data files is using *read.table* function.

```
> data3<-read.table("C:/Data/ex_1_comma.txt",
+                   header = FALSE, sep = ",")
> data3
```

```
  V1 V2 V3 V4 V5
1 10  M 80 84  A
2  7  M 85 89  A
3  4  F 90 86  B
4 20  M 82 85  B
5 25  F 94 94  A
6 14  F 88 84  C
```

Importing a Comma Separated Data

- The most convenient way to read in comma-separated data files is using *read.table* function.

```
> data3<-read.table("C:/Data/ex_1_comma.txt",
+                   header = FALSE, sep = ",")
> data3
```

```
  V1 V2 V3 V4 V5
1 10  M 80 84  A
2  7  M 85 89  A
3  4  F 90 86  B
4 20  M 82 85  B
5 25  F 94 94  A
6 14  F 88 84  C
```

- Equivalently, we can use *read.csv* function:

```
> data3<-read.csv("C:/Data/ex_1_comma.txt", header = FALSE)
```

Importing a Fixed Width Format File

The fixed format data file: `R_example_1_fixed.txt` has the following fixed format: subject(2), gender(1), CA1(3), CA2(3), HW(1).

```
1 10M 80 84A
2 7 M 85 89A
3 4 F 90 86B
4 20M 82 85B
5 25F 94 94A
6 14F 88 84C
```

- Use the function read.fwf - the widths of variables are specified in a vector.

```
> data3 <-read.fwf("C:/Data/ex_1_fixed.txt", width=c(2,1,3,3,1))
> data3
  V1 V2 V3 V4 V5
1 10  M 80 84  A
2  7  M 85 89  A
3  4  F 90 86  B
4 20  M 82 85  B
5 25  F 94 94  A
6 14  F 88 84  C
```

Importing Binary Files

- Binary data generated from other statistical software can be read into R (but it should be avoided).
- The R package *foreign* provides import facilities for some other statistical software.
- Activate the package by typing *library(foreign)*.

1. *read.spss(...)* reads in SPSS files.
2. *read.mtp(...)* imports Minitab worksheets.
3. *read.xport(...)* reads in SAS files in TRANSPORT format.
4. *read.S(...)* reads in binary objects produced by S-Plus.

To Assess a Dataframe

- Use *attach* command to make the variables in a dataframe accessible by their name within the R session, and use *names* command to get the list of the variable names.

```
> data3<-read.table("C:/Data/ex_1_name.txt", header = TRUE)
> data3
  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
2       7      M  85  89  A
3       4      F  90  86  B
4      20      M  82  85  B
5      25      F  94  94  A
6      14      F  88  84  C
> names(data3)
[1] "Subject" "Gender"  "CA1"     "CA2"     "HW"
> CA1
Error: object 'CA1' not found
>
```

To Assess a Dataframe

- Use *attach* command to make the variables in a dataframe accessible by their name within the R session, and use *names* command to get the list of the variable names.

```
> data3<-read.table("C:/Data/ex_1_name.txt", header = TRUE)
> data3
  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
2       7      M  85  89  A
3       4      F  90  86  B
4      20      M  82  85  B
5      25      F  94  94  A
6      14      F  88  84  C
> names(data3)
[1] "Subject" "Gender"  "CA1"     "CA2"     "HW"
> CA1
Error: object 'CA1' not found
>
```

- Need to use *attach* command:

```
> data3<-read.table("C:/Data/ex_1_name.txt", header = TRUE)
> attach(data3)
> CA1
[1] 80 85 90 82 94 88
```

Assessing Parts of a Dataframe: Column Subscript

Selecting some specified variables (columns):

```
> data3[,1]
[1] 10  7  4 20 25 14
> data3[,2:4]
  Gender CA1 CA2
1      M  80  84
2      M  85  89
3      F  90  86
4      M  82  85
5      F  94  94
6      F  88  84
```

Assessing Parts of a Dataframe: Row Subscript

Selecting some specified observations (rows):

```
> data3[1,]
  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
```

```
> data3[1:3,]
  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
2       7      M  85  89  A
3       4      F  90  86  B
```

Selecting some specific values in the data

```
> data3[3,3]
[1] 90
```

```
> data3[3,4]
[1] 86
```

Assessing Parts of a Dataframe: Logical Tests

```
> # all the rows (observations) whose gender = M:
> data3[Gender == "M",]

  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
2       7      M  85  89  A
4      20      M  82  85  B

> #all the rows (observations) whose gender = M and CA2>85
> data3[Gender == "M" & CA2 > 85,]

  Subject Gender CA1 CA2 HW
2       7      M  85  89  A
```

Assessing Parts of a Dataframe: Logical Tests (2)

Some logical operators in R:

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

Combining Dataframes by Rows (Observations)

```
> #create a dataframe from first 3 rows of data3:
> data1<-data.frame(data3[1:3,])
> #create a dataframe from last 3 rows of data3:
> data2<-data.frame(data3[4:6,])
> data<-rbind(data1,data2)
> #Combining two dataframes above:
> data
```

	Subject	Gender	CA1	CA2	HW
1	10	M	80	84	A
2	7	M	85	89	A
3	4	F	90	86	B
4	20	M	82	85	B
5	25	F	94	94	A
6	14	F	88	84	C

If the variables (name of columns) are not the same in both dataframes, an error message will be displayed.

Combining Dataframes by Variables (Columns)

```
> #create a dataframe from first 3 columns of data3:
> data1<-data.frame(data3[,1:3])
> #create a dataframe from the last two columns of data3:
> data2<-data.frame(data3[,4:5])
> #Combining two dataframes above:
> data<-cbind(data1,data2)
> data
```

```
  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
2       7      M  85  89  A
3       4      F  90  86  B
4      20      M  82  85  B
5      25      F  94  94  A
6      14      F  88  84  C
```

```
> #If data2<-data.frame(data3[,4]) then how the data looks like?
```

If the variables (name of columns) do not have the same length in in both dataframes, an error message will be displayed.

Combining Dataframes by Variables (1)

- Using *merge* command to merge 2 dataframes by a common variable.

```
> data1<-read.table("C:/Data/ex_1_IQ.txt", header = TRUE)
> data1
  Subject  IQ
1      10 106
2       7 112
3       4 119
4      20 102
5      25 125
6      14 101
> data<-merge(data3,data1 ,by="Subject", all=TRUE)
> data
  Subject Gender CA1 CA2 HW  IQ
1       4      F  90  86  B 119
2       7      M  85  89  A 112
3      10      M  80  84  A 106
4      14      F  88  84  C 101
5      20      M  82  85  B 102
6      25      F  94  94  A 125
```

Combining Dataframes by Variables (2)

- Equivalently, we can use:

```
> attach(data1)

The following object is masked from data3:

    Subject

> data<-data.frame(data3, IQ)
> data
  Subject Gender CA1 CA2 HW  IQ
1      10      M  80  84  A 106
2       7      M  85  89  A 112
3       4      F  90  86  B 119
4      20      M  82  85  B 102
5      25      F  94  94  A 125
6      14      F  88  84  C 101
```

Sorting a Dataframe

- Sort data3 by CA1 in ascending order:

```
> data3[order(CA1),]
  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
4      20      M  82  85  B
2       7      M  85  89  A
6      14      F  88  84  C
3       4      F  90  86  B
5      25      F  94  94  A
```

Sorting a Dataframe

- Sort data3 by CA1 in ascending order:

```
> data3[order(CA1),]
  Subject Gender CA1 CA2 HW
1      10      M  80  84  A
4      20      M  82  85  B
2       7      M  85  89  A
6      14      F  88  84  C
3       4      F  90  86  B
5      25      F  94  94  A
```

- Sort data3 by CA2 in descending order and only variables Subject, CA1 and CA2 are selected:

```
> data3[rev(order(CA2)), c(1,3:4)]
  Subject CA1 CA2
5      25  94  94
2       7  85  89
3       4  90  86
4      20  82  85
6      14  88  84
1      10  80  84
> #data3[rev(order(CA2)), c("Subject","CA1","CA2")]
```

while Loop

- The `while` loop is in the form of `while (condition) {expression}`

while Loop

- The `while` loop is in the form of `while (condition) {expression}`
- How this loop works:

while Loop

- The `while` loop is in the form of `while (condition) {expression}`

- How this loop works:
  1. `(condition)` must evaluate to a TRUE or a FALSE.

while Loop

- The `while` loop is in the form of `while (condition) {expression}`

- How this loop works:
  1. `(condition)` must evaluate to a TRUE or a FALSE.
  2. If `(condition)` is TRUE, do all the steps inside the while code block

while Loop

- The `while` loop is in the form of `while (condition) {expression}`

- How this loop works:
  1. `(condition)` must evaluate to a TRUE or a FALSE.
  2. If `(condition)` is TRUE, do all the steps inside the while code block
  3. Check `(condition)` again

while Loop

- The `while` loop is in the form of `while (condition) {expression}`

- How this loop works:
  1. `(condition)` must evaluate to a TRUE or a FALSE.
  2. If `(condition)` is TRUE, do all the steps inside the while code block
  3. Check `(condition)` again
  4. Repeat until `(condition)` is a FALSE.

while Loop: Examples

- Find the sum of first 10 integers:

```
> x<-0; S<-0
> while(x<=10) {S<- S+ x
+                  x<-x+1}
> S
[1] 55
```

while Loop: Examples

- Find the sum of first 10 integers:

```
> x<-0; S<-0
> while(x<=10) {S<- S+ x
+                  x<-x+1}
> S
[1] 55
```

- To print all the squares of integers from 1 to 5:

```
> x <- 0
> test <- TRUE  # or: test <- 1
> while(test>0){x <- x+1
+ test <- isTRUE(x<6) # or (x<6)
+ cat(x^2,test,"\n") }
1 TRUE
4 TRUE
9 TRUE
16 TRUE
25 TRUE
36 FALSE
```

`for` Loop (1)

- The `for` loop is in the form of:
  `for (<variable> in <range>) {expression}`

`for` Loop (1)

- The `for` loop is in the form of:
  `for (<variable> in <range>) {expression}`

- How this loop works:

`for` Loop (1)

- The `for` loop is in the form of:
  `for (<variable> in <range>) {expression}`

- How this loop works:
  1. Each time through the loop, `<variable>` takes a value

`for` Loop (1)

- The `for` loop is in the form of:
  `for (<variable> in <range>) {expression}`

- How this loop works:
  1. Each time through the loop, `<variable>` takes a value
  2. First time, `<variable>` starts at the smallest value in the range and do all the steps inside the `{expression}`.

`for` Loop (1)

- The `for` loop is in the form of:
  `for (<variable> in <range>) {expression}`

- How this loop works:
  1. Each time through the loop, `<variable>` takes a value
  2. First time, `<variable>` starts at the smallest value in the range and do all the steps inside the `{expression}`.
  3. Next time, `<variable>` gets the previous value + 1, until the last value in the range.

for Loop (2)

Example: find the sum of first 10 integers:

- ```
  > S<-0; for(i in 1:10){S <-S+i}
  > S
  [1] 55
  ```
  ```

for Loop (2)

Example: find the sum of first 10 integers:

- ```
  > S<-0; for(i in 1:10){S <-S+i}
  > S
  [1] 55
  ```
- A more advanced way:

```
> x <- numeric(10)
> for (i in 1:10){s <- 0
+                 for (j in 1:i){s <- s+j}; x[i] <- s
+ cat("The sum of the first ", i, "numbers = ", x[i], "\n")}
The sum of the first  1 numbers =  1
The sum of the first  2 numbers =  3
The sum of the first  3 numbers =  6
The sum of the first  4 numbers =  10
The sum of the first  5 numbers =  15
The sum of the first  6 numbers =  21
The sum of the first  7 numbers =  28
The sum of the first  8 numbers =  36
The sum of the first  9 numbers =  45
The sum of the first  10 numbers =  55
```
```

Output: *sink* Function

```
>
> sink("C:/Data/datasink_ex1.txt")
> x <- numeric(15)
> for (i in 1:15)
+ { s <- 0
+ for (j in 1:i) {s <- s+j}
+ x[i] <- s
+ cat("The sum of the first ", i, "numbers = ",x[i], "\n")
+ }
> sink()
> cat("The sum of the first ", i, "numbers = ",x[i], "\n")
The sum of the first  15 numbers =  120
>
```

All the output are stored in a text file named `datasink_ex1` under directory "C:/Data".

- The *sink* function is used to send objects and text to a file.
- This is useful when we want to keep a copy of the output in a file or when the contents of an object or function that may be too big to display on screen.

Output: *cat* Function

- The *cat* functions prints the output shown in the console unless redirected by sink.

```
> cat("The sum of the first ", i, "numbers = ",x[i], "\n")
The sum of the first  10 numbers =  55
```

Output: *cat* Function

- The *cat* functions prints the output shown in the console unless redirected by sink.

```
> cat("The sum of the first ", i, "numbers = ",x[i], "\n")
The sum of the first  10 numbers =  55
```

- "\n" is used to tell R to start a new line after this point.

write.table and *write.csv* Function

- The function *write.table* or *write.csv* can be used to write dataframes to a file.

```
> data
  Subject Gender CA1 CA2 HW  IQ
1      10      M  80  84  A 106
2       7      M  85  89  A 112
3       4      F  90  86  B 119
4      20      M  82  85  B 102
5      25      F  94  94  A 125
6      14      F  88  84  C 101
> write.table(data,"C:/Data/ex_1_with_IQ.txt")
```

- Dataframe "data" will be written to a text file (.txt) named "ex_1_with_IQ" in the directory `C:/Data`.

Built-in or Existing Functions in R

x and y are vectors. Some functions on vector in R:

- max(x): maximum value of x
- min(x): minimum value of x
- sum(x): total of all the values in x
- mean(x): arithmetic average values in x
- median(x): median value of x
- range(x): min(x), max(x)
- var(x): sample variance of x, with degrees of freedom = length(x) - 1
- cor(x,y): correlation between vectors x and y
- sort(x): a sorted version of x
- rank(x): vector of the ranks containing the permutation to sort x into ascending order

User-define Functions

- Characteristics of a function:
  -has a name
  -has parameters (0 or more)
  -has a body
  -returns something

User-define Functions

- Characteristics of a function:
  -has a name
  -has parameters (0 or more)
  -has a body
  -returns something

- How to write/define:
  `name <-` function`(parameters) { function body }`

User-define Functions

- Characteristics of a function:
  -has a name
  -has parameters (0 or more)
  -has a body
  -returns something

- How to write/define:
  `name <-` function`(parameters) { function body }`

- At the end of the `function body`, some command to ask for evaluation and return should be included.

User-define Functions: Examples

- A function to find sum of all the values in a vector:

```
> int<-c(1:10)
> s=function(x)sum(x)
> s(int)
[1] 55
```

User-define Functions: Examples

- A function to find sum of all the values in a vector:

```
> int<-c(1:10)
> s=function(x)sum(x)
> s(int)
[1] 55
```

- A function to find the standard error of the mean of a vector (SE of $\bar{x}$):

```
> se<-function(x){sqrt(var(x)/length(x))}
> # the command inside {} asks for a return
> se(CA1)
[1] 2.125245
```

User-define Functions: Examples

- A function to find sum of all the values in a vector:

```
> int<-c(1:10)
> s=function(x)sum(x)
> s(int)
[1] 55
```

- A function to find the standard error of the mean of a vector (SE of $\bar{x}$):

```
> se<-function(x){sqrt(var(x)/length(x))}
> # the command inside {} asks for a return
> se(CA1)
[1] 2.125245
```

- *if () else* in R: *if* statement can be followed by an optional *else* statement which executes when the boolean expression is false.

```
> x<-0
> for (i in 1:7){if (i<=5){x<-x+i}else{print("STOP")}}
[1] "STOP"
[1] "STOP"
> x
[1] 15
```

User-define Functions: Examples

- A function to find sum of all the values in a vector:

```
> int<-c(1:10)
> s=function(x)sum(x)
> s(int)
[1] 55
```

- A function to find the standard error of the mean of a vector (SE of $\bar{x}$):

```
> se<-function(x){sqrt(var(x)/length(x))}
> # the command inside {} asks for a return
> se(CA1)
[1] 2.125245
```

- *if () else* in R: *if* statement can be followed by an optional *else* statement which executes when the boolean expression is false.

```
> x<-0
> for (i in 1:7){if (i<=5){x<-x+i}else{print("STOP")}}
[1] "STOP"
[1] "STOP"
> x
[1] 15
```

- Question: write a function to find the median of a given dataset.

Further Reading

- A good source for reading more details about building functions in R: `https://datasciencebeginners.com/2018/11/02/10-user-defined-functions-in-r/`