# R Programming

Vik Gopal

*After three days without programming,*
*life becomes meaningless.*

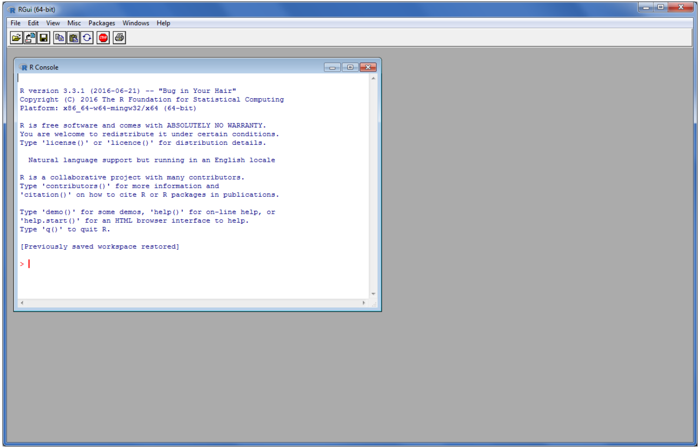
1. Installing R and RStudio
2. R Objects
3. R Syntax
4. Conditional Execution and Loops
5. Functions
   - In-built R Functions
   - Writing R Functions
   - The `apply` Family of Functions (also in-built)
6. Important Classes of Objects
   - Strings
   - Factors
   - Date Objects
7. Basic Plotting With R
   - Line and Scatterplots
   - Barcharts
8. Generating Reports With R Markdown

# Installing R

- To download R, go to CRAN, the Comprehensive R Archive Network.
  - The URL is `https://cran.r-project.org/`
- Download the version for your operating system and install it.
- A new major version is released once a year, and there are 2 - 3 minor releases each year.
- Upgrading is painful, but it gets worse if you wait to upgrade.
- Please ensure that you have version 4.1 or later for our class. Functions in older versions work differently, so you might face problems.

# R Graphical User Interface (GUI)

After installing, you can start using R straightaway:




However, the basic GUI is not very user-friendly.

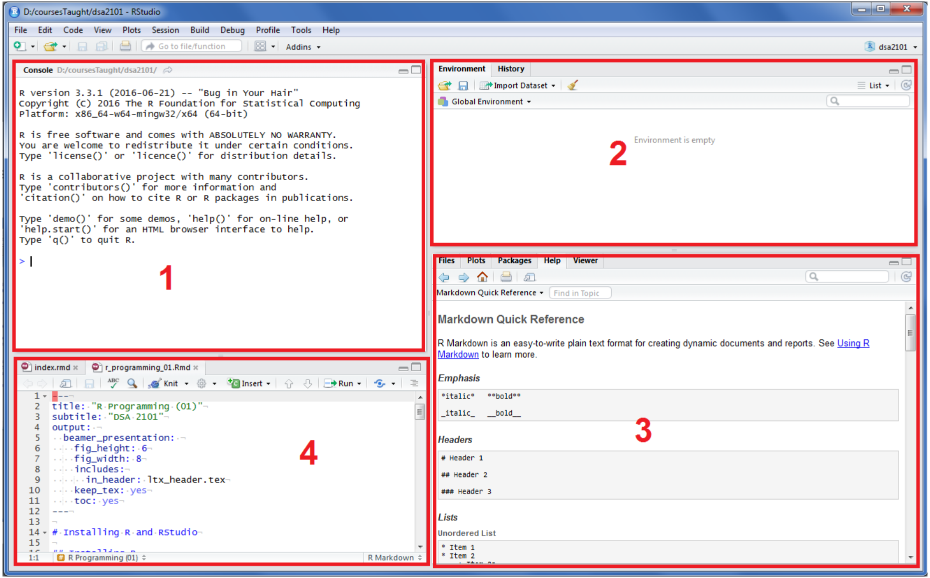
# An IDE for R

- Instead of using the basic GUI for R, we are going to use RStudio.
- RStudio is an Integrated Development Environment (IDE) for R.
- It provides several features that base R does not:
  - A history of previous plots made.
  - Integration with markdown. We shall learn about this soon. It is a way of integrating R source code and output into a single document for distribution.
  - The ability to browse the objects in our workspace more easily.
- RStudio:R is analagous to Eclipse:Java, Spyder:Python, etc.

# Installing RStudio

- The installation file for RStudio can be obtained from this URL:
  - https://www.rstudio.com/products/rstudio/download/
- It is updated a couple of times a year.
- Make sure you have at least version 2022.07.x for our course.

# RStudio Interface



# RStudio Panels

panels 1 and 2

1. Panel 1 is the **console**.
   - This is where you type R commands.
   - The output from these commands or functions will also be seen here.
   - Use the $\uparrow$ key to scroll through previously entered commands.
2. Panel 2 contains the **History** and **Environment** tabs.
   - The **History** tab displays all commands that have been previously entered in the current session.
   - These commands can be sent directly to the source code panel or the console panel.
   - The **Environment** tab in this panel has a list of items that have been created in the current session.

# RStudio Panels

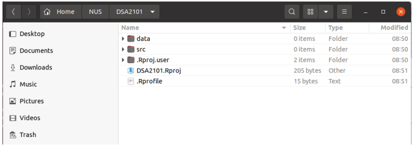
panels 3 and 4

3. Panel 3 contains the **Files**, **Plots** and **Help** tabs.
    - The **Files** tab contains a directory structure that allows one to choose and open files in the source code editor.
    - Through the **Plots** tab, one can access all plots that have been created in the current session.
    - The **Help** tab displays the documentation for R functions.
4. Panel 4 contains the source code editor.
    - This where you edit R scripts.
    - You can hit *Ctrl-Enter* to execute a command (in the console panel) while your cursor is in the source code panel.
    - You can also highlight code and the editor and execute it directly in the console panel.

# RStudio Projects

- Similar to other IDEs, RStudio allows you to create projects in order to keep different analyses separate.
- I strongly recommend the following practice:
  - Create a folder *DSA2101* for our class, and create a new RStudio project there.
  - Add `.Rprofile` to this level.
  - Within DSA2101, create a folder called `src/` to store all your source codes and Rmd files.
  - Within DSA2101, create a folder called `data/` to store all data files.
  - Thus `src/` and `data/` should be at the "same level".

## Directory Structure




## Contents of .Rprofile

```
setwd("./src")
```

# The Working Directory

- The working directory is where R looks when you ask it to read from a file, or to write to a file.
- You can get and set it with

```
getwd()
#setwd()
```

- Sometimes, a file that you want to read is not in the working directory, but in the directory next to it. In such situations, you can tell R to "go up one directory and then down into the next folder", or you can give R the full path to the file. Here are two examples:

```
list.files("../data/")
list.files("/home/viknesh/NUS/coursesTaught/dsa2101/src")
```

- **Use relative paths in all code you write.** Failure to do so makes makes it harder for you to share your code with others.

# The ‘?’ Operator

- If you need to find out more information about a particular function e.g. what arguments it expects, what it does, and so on, use the following command:

```
?mean
```

- If you are not sure about the name of the function, you can use the following syntax:

```
??mean
```

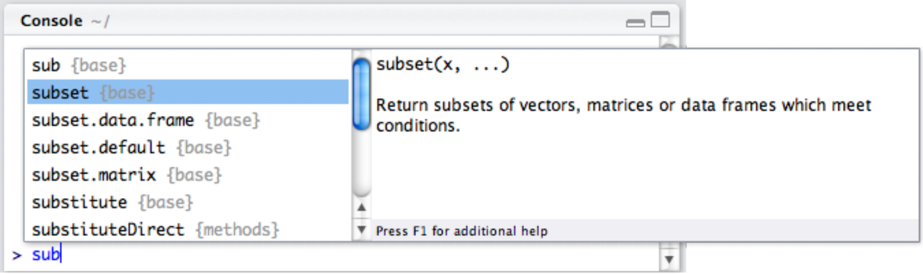
This command will return a list of search results based on the word `mean`.

# The Help Pages

- Documentation pages are (usually) very thorough:
  - They begin with a description of all possible arguments that can be provided to a function.
  - Next, the page describes what the function does.
  - Following this, there will be references and related commands.
  - Right at the bottom will be a list of examples of how to call this particular function.
- It is **crucial** to be comfortable reading R documentation.

# Code Completion Feature

- When in the source code editor or the console panel, RStudio will be able to autocomplete the function or argument that you are typing.
- For example, type `sub` and hit the *Tab* key. You will get a list of suggested commands, that all begin with `sub`



# R Packages

- There are numerous packages that can be installed to extend the functionality of R. Most of these are hosted on CRAN (Comprehensive R Archive Network):
https://cran.r-project.org/web/packages/available_packages_by_name.html
    - At last check (July 2022), there were over 18,000 packages.
- To install one of these packages, you can use:

```
# installs stringr package, and its dependencies.
install.packages("stringr")
```

- Before using the functions in a package, we have to load it:

```
library(stringr)
```

- To access a list of all available functions from a package, use:

```
help(package="stringr")
```

1. Installing R and RStudio
2. R Objects
3. R Syntax
4. Conditional Execution and Loops
5. Functions
   - In-built R Functions
   - Writing R Functions
   - The `apply` Family of Functions (also in-built)
6. Important Classes of Objects
   - Strings
   - Factors
   - Date Objects
7. Basic Plotting With R
   - Line and Scatterplots
   - Barcharts
8. Generating Reports With R Markdown

# Object Oriented

- When working in R, it useful to remember these two principles:
  1. Everything in R is an object.
  2. Every object has a class.
- If we know the class of an object, we can begin to understand what functions can be applied to it, and in general how it will behave in different conditions.

# Naming Objects in R

- In the course of working in R, it is necessary to create new R objects, either
  - to store data, or
  - to store the output of a function.
- When doing so, the use of the following words/letters as names should be avoided as they are reserved by R:

```
FALSE Inf NA NaN NULL TRUE
break else for function if in next repeat while
c q s t C D F I T
```

- Also note that R is case sensitive, so `Alfred` and `alfred` refer to different objects within R.

# Vectors

- R has no scalars. The basic building block for storing data is a *vector*.
- The first line below creates a vector of length 3, containing the values $(1, 2, 3)$. The vector is called Z.
- <- is the assignment operator in R. It is used to assign names to objects.
- The second line prints Z.

```
Z <- c(1,2,3)
Z
```

```
# Compare to Python syntax:
Z = [1,2,3]
```

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

- Now, your workspace contains the object Z. Go to panel 2 of RStudio and click on the **Environment** tab. You should see it there.

# Creating Vectors

- Vectors can be created using many different commands.
- The `:` in the first line below, tells R to create a sequence of integers starting from -2 and ending at 2. We shall learn a more efficient and general method in a later section.
- The third input line tells R to use the vector `X` to take powers of 2, i.e. to compute $2^{-2}, 2^{-1}, 2^0, 2^1$ and $2^2$.
  - This syntax is an example of the *recycling rule* in R.

```
-2:2                    # creates an evenly spaced sequence.
```

```
[1] -2 -1  0  1  2
```

```
X <- -2:2               # assign name X to sequence
Y <- 2^X                # raise 2 to powers given by X
Y                       # print Y
```

```
[1] 0.25 0.50 1.00 2.00 4.00
```

# Accessing Elements Within a Vector

- Elements in a vector are accessed using a sequence of integers, and the `[ ]` parentheses.
- The first element of the vector is at position 1.
- The final line in the code below shows how to access the last element without knowing the length of the vector.

```
Y[2]            # Access element 2
```

```
[1] 0.5
```

```
Y[2:4]          # Access elements 2,3 and 4
```

```
[1] 0.5 1.0 2.0
```

```
Y[length(Y)]    # Access the last element
```

```
[1] 4
```

# Classes

Vectors can be of different classes. Here are the most fundamental classes in R:

1. **character** This means that each element of the vector is a character string.
2. **numeric** Each element of the vector is a real number.
3. **integer** Each element is an integer.
4. **logical** Each element is either a `TRUE` or a `FALSE`.
5. **factor** Each element is one of a few possible values. This is typically used to store categorical data. We shall see more about this soon.

# Vector Classes Examples

- The class of a vector is a property of the vector; we do not assign it.
- All elements of a vector will be of the same class.

```r
abc <- c("a", "b", "c")            # character
W <- c(1.2)                        # numeric, length 1
A <- c(TRUE, TRUE, FALSE, TRUE)   # logical
```

# Matrices

- Matrices are 2-dimensional arrays with all elements of the same type.

```
mymat <- matrix(1:9, nrow=3, ncol=3)
mymat        # try dim(mymat)
```

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

- Think of a matrix as a vector that is laid out in a special way.

```
mymat[4]
```

```
[1] 4
```

# Accessing Elements in a Matrix

- Elements in a matrix are accessed using the `[ , ]` parenthesis notation.
- For instance, the element in row *i* and column *j* of matrix `X` is retrieved with the syntax `X[i, j]`

```
mymat[3,2]
```

```
[1] 6
```

# Accessing Rows and Columns of a Matrix

- Row $i$ can be accessed using `X[i, ]`. Similarly, column $j$ can be accessed using `X[,j]`.

```r
mymat[2, ]
```

```
[1] 2 5 8
```

- A vector can be used in place of $i$ or $j$, which then returns a group of columns or rows.

```r
mymat[c(1,3), ]
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    3    6    9
```

# Accessing Named Rows and Columns

- It is possible to assign row and column names to a matrix, and then use these names to access them.

```
rownames(mymat) <- c("a", "b", "c")
colnames(mymat) <- c("i", "ii", "iii")
mymat["b","iii"]
```

```
[1] 8
```

```
mymat["b",c("i","iii")]
```

```
 i iii
 2   8
```

# Data Frames

- A data frame is the type of object normally used to store data in R.
- It is aligned with the concept of cases as rows, and variables or measurements in columns.
- The difference between a data frame and a matrix is that in a data frame, the columns could be of different classes.
    - In a matrix, all columns have to belong to the same class.

# Creating Data Frames

- Consider the following operating budget data for a company.

| Expenditure category | Amount |
|---|---|
| Manpower | \$519.4m |
| Asset | \$38.0m |
| Other | \$141.4m |

- Let us manually create a data frame in R, containing this data.

# Creating Data Frames

cont'd

- Here's how we can do it by hand:

```
exp_cat <- c("manpower", "asset", "other")
amount <- c(519.4, 38, 141.4)
op_budget <- data.frame(amount, exp_cat)
op_budget
```

```
  amount  exp_cat
1  519.4 manpower
2   38.0    asset
3  141.4    other
```

# Accessing Data Frames

- Data frames are *not* matrices, but their elements can be accessed using a similar syntax.

```r
op_budget[, "exp_cat"]
```

```
[1] manpower asset    other
```

```r
op_budget[c(3,2), ]
```

```
  amount exp_cat
3  141.4   other
2   38.0   asset
```

# Accessing Data Frames

cont'd

- In addition, the '$' symbol can be used to access individual columns in the data frame.

```
op_budget$amount
```

```
 [1] 519.4  38.0 141.4
```

```
op_budget$amount[1:2]
```

```
[1] 519.4  38.0
```

# Datasets in R

- There are several example datasets stored within R, as data frames. Let's inspect one of them.

```
class(cars)
```

```
[1] "data.frame"
```

```
names(cars)
```

```
[1] "speed" "dist"
```

```
head(cars, n=3)       # to see the first few rows.
```

```
  speed dist
1     4    2
2     4   10
3     7    4
```

# Lists

- A data frame is in fact a special type of structure within R known as a list.
- A list is simply a collection of objects. The objects can be different from one another.
- They can also be of different lengths; they can even be lists of their own!

# Creating and Accessing Lists

```
ls1 <- list(A=seq(1, 5, by=2), B=seq(1, 5, length=4))
ls1
```

```
$A
[1] 1 3 5

$B
[1] 1.000000 2.333333 3.666667 5.000000
```

```
ls1[[2]]
```

```
[1] 1.000000 2.333333 3.666667 5.000000
```

```
ls1[["B"]] # ls1$B is also OK
```

```
[1] 1.000000 2.333333 3.666667 5.000000
```

# Saving and Reading Back Objects

- The `saveRDS()` and `readRDS()` functions allow for an object to be saved and retrieved for use.

```
saveRDS(ls1, file="ls1.rds")
ls2 <- readRDS("ls1.rds")
ls2
```

```
$A
[1] 1 3 5

$B
[1] 1.000000 2.333333 3.666667 5.000000
```

# Alternatives to RDS

- The `save()` and `load()` functions allow us to save multiple objects into a file at one go.
- The advantage of using the RDS method instead of `save()` or `load()` is that the object can be renamed upon reading in.
- The disadvantage is that only one object can be stored in each file; this drawback can be easily overcome though, using a list.

# Removing Objects from the Workspace

- As you work, you might find that you accumulate a number of objects within your workspace.
- You can inspect them Environment tab, and remove them when in Grid view.
- You can also remove them with the R rm() function.

```
x <- 1:5
rm(list = c("x"))     # removes object named x
rm(list = ls())       # removes ALL objects! Be careful!
```

# Investigating the Structure of an Object

- Sometimes, we may be given an object without much information about it.
- The `str()` function can be very useful in inspecting it.
- The object in this example contains a list of hawker centres, retrieved from onemap.sg.

```
hawkers <- readRDS("../data/hawker_ctr_raw.rds")
class(hawkers)
```

```
[1] "list"
```

```
str(hawkers, max.level = 1)
```

```
List of 1
 $ SrchResults:List of 117
  .. [list output truncated]
```

# Hawkers Dataset

- Thus `hawkers` is a list of length 1. What's in that list?

```
# Try this:
str(hawkers[[1]], max.level = 1)
```

- Thus `hawkers` is a list of lists.
- The first sublist is of length 1.
- The remaining 116 are of length 12. What do they contain?

# Hawkers Dataset

cont'd

- Each of those lists contains information on a hawker centre.

```
str(hawkers[[1]][[2]], max.level=1)
```

```
List of 12
 $ ADDRESSBUILDINGNAME    : chr ""
 $ ADDRESSFLOORNUMBER     : chr ""
 $ ADDRESSPOSTALCODE      : chr "141001"
 $ ADDRESSSTREETNAME      : chr "Commonwealth Drive"
 $ ADDRESSUNITNUMBER      : chr ""
 $ DESCRIPTION            : chr "HUP Standard Upgrading"
 $ HYPERLINK              : chr ""
 $ NAME                   : chr "Blks 1A/ 2A/ 3A Commonwealth Drive"
 $ PHOTOURL               : chr ""
 $ ADDRESSBLOCKHOUSENUMBER: chr "1A/2A/3A"
 $ XY                     : chr "24055.5,31341.24"
 $ ICON_NAME              : chr "HC icons_Opt 8.jpg"
```

1 Installing R and RStudio

2 R Objects

3 R Syntax

4 Conditional Execution and Loops

5 Functions
- In-built R Functions
- Writing R Functions
- The `apply` Family of Functions (also in-built)

6 Important Classes of Objects
- Strings
- Factors
- Date Objects

7 Basic Plotting With R
- Line and Scatterplots
- Barcharts

8 Generating Reports With R Markdown

# Expresssions and Assignments

- In R, commands that we enter into the console are either expressions or assignments.
- An expression is evaluated and (normally) printed. For example,

```r
pi + 1
```

```
[1] 4.141593
```

- In an assignment, the expression portion of it is first evaluated, after which the output is passed to a variable. The result is not printed. For example,

```r
a <- 6 + 2
```

- The <- is the assignment operator. Although it can be replaced with =, the latter should be reserved for use within arguments of functions.

# Arithmetic Expressions

- The basic unit in R is a vector.
- Arithmetic operations are performed element by element.
- The following symbols represent the standard arithmetic operators of addition, subtraction, multiplication, division and exponentiation:

```
x + y
x - y
x * y
x / y
x ^ y
```

- For more arithmetic operations, type the following into your console:

```
?Arithmetic
```

# Vectorised Operations

- Most of R's functions are *vectorised*.
- This includes the operators on the previous slide. For example,

```r
x <- 5:10
y <- 10:15
x + y
```

```
[1] 15 17 19 21 23 25
```

# The Recycling Rule

- What happens when we do this?

```
x <- 5:10
x + 2
```

```
[1]  7  8  9 10 11 12
```

- Why should this happen? 2 is not a vector of length 6.
- R uses a recycling rule whenever it is presented with vectors of varying lengths in an expression.

# The Recycling Rule

cont'd

- The value of the expression is a vector with the same length as the longest vector in the expression.
- Shorter vectors are *recycled* as often as need, until they match the length of the longest vector.
- In particular, a single number is repeated an appropriate number of times.

# The Recycling Rule

cont'd

- Following on from the previous example,

```
x <- 5:10          # a vector of length 6
y <- c(x,x)        # a vector of length 12
2*x + y + 1        # a vector of length 12
```

```
[1] 16 19 22 25 28 31 16 19 22 25 28 31
```

# Logical Expressions

- Logical vectors are vectors of `TRUE` or `FALSE` values.
- These are most often generated by *conditions*.
- When the following binary operators are applied to numeric vectors, the output will be a logical vector:

```
x < y; x <= y ; x > y ; x >= y ; x == y ; x != y
```

# Examples of Logical Expressions

- Remind yourself that the recycling rule is in play for all of the above expressions.

```
x <- 1:5
x < 3
```

```
[1]  TRUE  TRUE FALSE FALSE FALSE
```

```
x == 1
```

```
[1]  TRUE FALSE FALSE FALSE FALSE
```

```
x != 17
```

```
[1] TRUE TRUE TRUE TRUE TRUE
```

# Logical Expressions

cont'd

- It is also possible to find the intersection, union and negation of vector-valued logical expressions using | and &.

```
y <- x <= 3
z <- x >= 3
y & z
```

```
[1] FALSE FALSE  TRUE FALSE FALSE
```

```
!y
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE
```

# Logical Expressions

cont'd

- The & operator takes the elementwise AND operation of two vectors.
- The | operator takes the elementwise OR operation of two vectors.
- The `!` operator takes the elementwise NOT operation of a vector.

# Selection with Logical Expressions

- Logical vectors can be used to select a subset of elements of a vector.

```
x
```

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

```
y
```

```
[1]  TRUE  TRUE  TRUE FALSE FALSE
```

```
x[y]
```

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

- We shall return to this particular use of logical vectors when learning about subsetting data.

1 Installing R and RStudio

2 R Objects

3 R Syntax

4 Conditional Execution and Loops

5 Functions
- In-built R Functions
- Writing R Functions
- The `apply` Family of Functions (also in-built)

6 Important Classes of Objects
- Strings
- Factors
- Date Objects

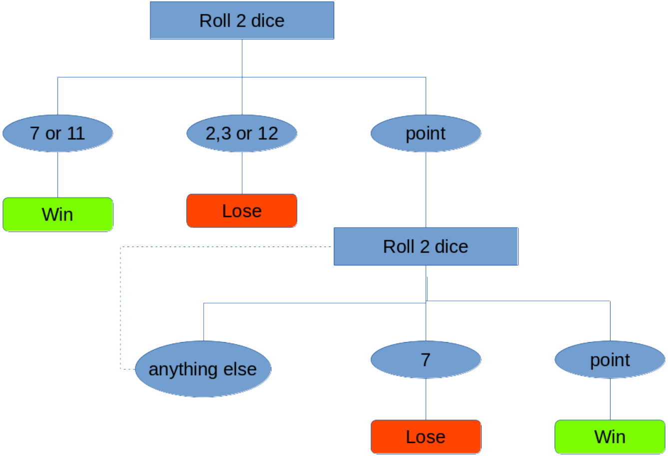
7 Basic Plotting With R
- Line and Scatterplots
- Barcharts

8 Generating Reports With R Markdown

# Conditional Executions

- Many times, we wish to execute a certain set of instructions if a condition is TRUE, and another set otherwise.
- In these cases, we use `if-else` statements.
- Suppose that we wish to simulate a game of craps.
- Craps is a casino game, played with two dice.
  - https://en.wikipedia.org/wiki/Craps
- It is in fact quite complicated in terms of the bets you can place, but here are the basic rules.

# Craps Flowchart



# Craps Flowchart

cont'd

1. Begin by rolling 2 dice.
   - If you obtain a 7 or 11, you **win**.
   - If you obtain 2, 3, or 12, you **lose**.
   - Any other number is termed the "point".
2. If you are "on point", you continue rolling both dice until one of the following happens:
   - you obtain point once more, in which case you **win**.
   - you obtain 7, in which case you **lose**.

You should be able to calculate that the probability of winning is

$$244/495 \approx 49.3\%$$

# Rolling a Die

- The following code will simulate the roll of a single six-sided die.

```
sample(1:6, size=1)
```

- The code will select a single observation from the vector $1, 2, 3, 4, 5, 6$, with all observations having equal probability of being chosen.
- If we wish to roll **two** dice, then we need the following code.

```
sample(1:6, size=2, replace=TRUE)
```

- The extra argument will instruct R to carry out sampling with replacement.

# Code for Craps

```
die_values <- sample(1:6, size=2, replace=TRUE)
total_val <- sum(die_values)

if(total_val %in% c(7, 11)){
  outcome <- "win"
} else if (total_val %in% c(2, 3, 12)) {
  outcome <- "loss"
} else {
  point <- total_val
  total_val <- 0
  repeat {
    die_values <- sample(1:6, size=2, replace=TRUE)
    total_val <- sum(die_values)

    if(total_val == 7){
      outcome <- "loss"
      break
    } else if (total_val == point) {
      outcome <- "win"
      break
    }
  }
}
```

# Code for Craps

cont'd

- Observe the use of the following operators:
  - `%in%`
  - `repeat`
  - `break`
- We could also have used a `while` loop in place of the `repeat` block.

# Repeated Executions

- Now suppose we wish to simulate the game for 1000 iterations.
- For such loops, we could use a `for` loop. Later we shall learn about how special `R` functions to loop functions, but for now, let's understand how a `for` loop is written in R.
- At every iteration, we store the result in a character vector `res1`.

# 10000 Games of Craps

```
set.seed(2102)                          # for reproducibility
res1 <- rep("a", 10000)                 # instantiate results vector

for(i in seq_along(res1)) {             # 'for' loop begins here
  # Craps code from previous slide here
  # ...
  # ...
  res1[i] <- outcome
}
table(res1)
```

```
table(res1)
loss  win
5180 4820
```

The proportion of times that a win results is 0.482. Does this agree with theory?

1 Installing R and RStudio

2 R Objects

3 R Syntax

4 Conditional Execution and Loops

5 Functions
- In-built R Functions
- Writing R Functions
- The `apply` Family of Functions (also in-built)

6 Important Classes of Objects
- Strings
- Factors
- Date Objects

7 Basic Plotting With R
- Line and Scatterplots
- Barcharts

8 Generating Reports With R Markdown

# Function Arguments

- We have already encountered a few functions in R. As you have noticed, they can take arguments that modify their behaviour.
- Not all of them need to be specified.
- The `args()` function can be used to list the arguments of a function.

```
args(plot.default)
```

```
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
    ann = par("ann"), axes = TRUE, frame.plot = axes,
    panel.first = NULL, panel.last = NULL, asp = NA, ...)
NULL
```

- `type` and `main` have default values.
- `x` does not. Its value **must** be supplied.
- `...` refers to additional arguments that can be supplied, and will be passed on to other functions

1. Installing R and RStudio
2. R Objects
3. R Syntax
4. Conditional Execution and Loops
5. Functions
   - In-built R Functions
   - Writing R Functions
   - The `apply` Family of Functions (also in-built)
6. Important Classes of Objects
   - Strings
   - Factors
   - Date Objects
7. Basic Plotting With R
   - Line and Scatterplots
   - Barcharts
8. Generating Reports With R Markdown

# Function seq()

- The seq() command is used to generate regular sequences. It is more general than the ':' operator that we have seen. It can be called in several ways. Here are two common ways.
  - Generate a sequence starting *from* a certain value, up *to* a certain value, with elements in the sequence separated *by* a certain amount.
  - Generate a sequence starting *from* a certain value, up *to* a certain value, with the final sequence having a particular *length*.

```
# Sequence using by:
seq(from=1, to=2, by=0.2)
```

```
[1] 1.0 1.2 1.4 1.6 1.8 2.0
```

```
# Sequence using length:
seq(from=1, to=2, length=4)
```

```
[1] 1.000000 1.333333 1.666667 2.000000
```

# Function rep()

- The `rep()` function is used to repeat a sequence. We can either:
  - Repeat a sequence a specified number of *times*
  - Repeat each element in a sequence a specified number of *times*.

```
x <- seq(1, 2, length=3)
rep(x, times=3)
```

```
 [1] 1.0 1.5 2.0 1.0 1.5 2.0 1.0 1.5 2.0
```

```
rep(x, times=c(1,2,3))
```

```
 [1] 1.0 1.5 1.5 2.0 2.0 2.0
```

# Function `paste()`

- The `paste()` command is used to concatenate vectors, after converting them to characters.
- It is very commonly used when we wish to create labels, either in a plot or in a dataset.

```
paste("A", 1:6, sep = "")
```

```
[1] "A1" "A2" "A3" "A4" "A5" "A6"
```

- Take note of the recycling rule in play here:

```
paste(c("A","B"), 1:3, sep = "")
```

```
[1] "A1" "B2" "A3"
```

1. Installing R and RStudio
2. R Objects
3. R Syntax
4. Conditional Execution and Loops
5. Functions
   - In-built R Functions
   - Writing R Functions
   - The `apply` Family of Functions (also in-built)
6. Important Classes of Objects
   - Strings
   - Factors
   - Date Objects
7. Basic Plotting With R
   - Line and Scatterplots
   - Barcharts
8. Generating Reports With R Markdown

# Writing Your Own Functions

- At some point, we will have to write a function of our own.
- We shall have to decide
  - what arguments it should take,
  - whether these arguments should have defaults, and if so, what those default values should be.
  - what that function should return.
- The typical approach is to write a sequence of expressions that work, then package them into a function.
- Let's suppose that we wish to write a function to simulate one game of dice between A and B.

# Craps Function

```
craps_game <- function() {
  die_values <- sample(1:6, size=2, replace=TRUE)
  total_val <- sum(die_values)

  if(total_val %in% c(7, 11)){
    return("win")
  } else if (total_val %in% c(2, 3, 12)) {
    return("loss")
  } else {
    point <- total_val
  }
  total_val <- 0
  repeat {
    die_values <- sample(1:6, size=2, replace=TRUE)
    total_val <- sum(die_values)

    if(total_val == 7){
      return("loss")
    } else if (total_val == point) {
      return("win")
    }
  }
}
```

# Clearer Code

- The '`for`' loop on slide 62 is much easier to read now:

```
set.seed(2102)

res1 <- rep("a", 10000)

for(i in seq_along(res1)) {
  res1[i] <- craps_game()
}
```

# Debugging

- When we write a function, we seldom get it right the first time.
- Sometimes, it fails with an error message or warning from R. Other times, it runs to completion, but gives us dubious results.
- In such situations, we have to debug our function. We can do this in 3 ways:
  - By inserting print statements in the function to keep track.
  - By inserting a breakpoint in the function, then stepping through from that point on. Breakpoints can be inserted with the `browser()` statement.
  - By stepping through the function from start till finish.

# Debugging From Start To Finish

- To debug a function from its first line, run the following code before your function:

```
debug(craps_game)
craps_game()
```

- When you have fixed the error and wish to return to normal execution of the function, do this:

```
undebug(craps_game)
```

1. Installing R and RStudio
2. R Objects
3. R Syntax
4. Conditional Execution and Loops
5. Functions
    - In-built R Functions
    - Writing R Functions
    - The `apply` Family of Functions (also in-built)
6. Important Classes of Objects
    - Strings
    - Factors
    - Date Objects
7. Basic Plotting With R
    - Line and Scatterplots
    - Barcharts
8. Generating Reports With R Markdown

# Repeated Application of Functions

- In data analysis, we find ourselves having to repeat the same operation several times.
- For instance, we may need to split a dataset by age-group, and then take the mean height for each group.
- Or we might have a matrix of values, and we need to take the mean of each column or row.
- The `apply` family of functions allow us to repeatedly apply a function. We shall cover:
  - `apply()`
  - `sapply()`, and
  - `lapply()`

# The `apply()` Function

- If we have a matrix, and we wish to apply a function to each row or column separately, then the `apply()` function is what we need.
- The following code generates a 100 x 2 matrix with $N(2, \ \sigma^2 = 4)$ random variables.

```
set.seed(2105)
X <- rnorm(200, mean=2, sd=2)
X <- matrix(X, nrow=100)
head(X, n=2)
```

```
          [,1]     [,2]
[1,] 0.6226833 3.779411
[2,] 4.1525270 2.342434
```

# The `apply()` Function

cont'd

- The following code applies (`apply`) the function mean (`mean`) to each column (2) of a matrix (`X`), thus computing column means.

```
col_means <- apply(X, 2, mean)
col_means
```

```
[1] 1.820889 2.084210
```

- The following code applies (`apply`) the function mean (`mean`) to each row (1) of a matrix (`X`), thus computing row means.

```
row_means <- apply(X, 1, mean)
head(round(row_means, digits = 2))
```

```
[1] 2.20 3.25 1.16 2.93 3.01 1.88
```

# The `apply()` Function

cont'd

- The `mean()` function has additional arguments.
- One of them specifies that outliers be removed before the mean is computed.
  - The argument is `trim`.
- This additional argument can be supplied in the `apply()` call as well:

```
trimmed_col_means <- apply(X, 2, mean, trim=0.1)
trimmed_col_means
```

```
[1] 1.742469 2.151429
```

# Anonymous Functions

- Instead of using pre-defined functions to apply to each row/column, we can define functions on-the-fly.
- Suppose we wished to count the proportion of values greater than 0 in each column:

```
apply(X, 2, function(x) sum(x > 0)/nrow(X))
```

```
[1] 0.81 0.85
```

# Not A Matrix

- What if the object that we wish to iterate over is not the row or column of a matrix?
- In such cases, we use `sapply()` or `lapply()`.
  - `sapply()` returns a vector or a matrix,
  - `lapply()` returns a list, which is necessary if the output of each function call is not a vector of the same length.

# `sapply()` With Hawker Centre Data.

- Let's recall the hawker centre data.
- Suppose that we need to extract the street name of each hawker centre in the dataset.
- Recall that we have a list of length 116, where each component is in itself a list.

```
hawk_116 <- hawkers[[1]][-1]
str(hawk_116, max.level = 1)
```

```
List of 116
 $ :List of 12
 $ :List of 12
 $ :List of 12
 $ :List of 12
  [list output truncated]
```

# Retrieving Street Names

- First we experiment with one of the individual components to get it right, then hit it with `sapply()`.

```
hawk_116[[2]]$ADDRESSSTREETNAME
```

```
[1] "Marsiling Lane"
```

```
street_names <- sapply(hawk_116, function(x) x$ADDRESSSTREETNAME)
head(street_names)
```

```
[1] "Commonwealth Drive" "Marsiling Lane"     "Boon Lay Place"
[4] "Havelock Road"      "Circuit Road"       "Whampoa Drive"
```

# Replacing a for Loop

- Recall the example where we generated 10000 iterations of a game of craps.
- Here's how we can do it with `sapply()`:

```
set.seed(2102)
game_results <- sapply(1:10000, function(x) craps_game())
table(game_results)
```

```
game_results
loss  win
5180 4820
```

- The anonymous function iterates over the values $1, 2, 3, \ldots, 1000$ but does not actually use them!
  - We have tricked R into repeated application of a function that we already had.

# The `lapply` Function

- The `lapply()` is very similar to `sapply()`, except that it returns a list instead of a vector.
- It is most useful when the output of the repeated function may not be all of the same length/type.

```
set.seed(2106)
x <- list(A = rnorm(10, mean=1), B= rnorm(10, mean=0))
lapply(x, function(y) y[y<0.5])
```

```
$A
[1]  0.28023059 -0.30439802  0.02989543

$B
[1] -0.2192394 -1.5862490  0.3457230 -2.2485464 -0.1630706
```

# When to use `s/lapply`, `for`, or Neither

Here are some general guidelines on when to use these functions.

- If you are repeating a function of your own, you probably need to call `sapply` or `lapply`.
- If you are repeating a base R function over a vector, you should not need either `for` or `s/lapply`, since these functions are already vectorised.
- These are both poor practices in R:

```r
X <- sample(100, size=50)
logX <- sapply(X, log)
```

```r
X <- sample(100, size=50)
for(i in 1:50) {
  logX[i] <- log(X[i])
}
```

- This is the correct way in `R`:

```r
logX <- log(X)
```

- I usually use a for loop when:
  - there are many statements, and I only need to execute the loop once.
  - the output from a previous iteration is required for subsequent iterations.

1 Installing R and RStudio

2 R Objects

3 R Syntax

4 Conditional Execution and Loops

5 Functions
- In-built R Functions
- Writing R Functions
- The `apply` Family of Functions (also in-built)

6 Important Classes of Objects
- Strings
- Factors
- Date Objects

7 Basic Plotting With R
- Line and Scatterplots
- Barcharts

8 Generating Reports With R Markdown

# Prerequisites

- This section will focus on string manipulations in R.
- We shall use the `stringr` package, since it is more powerful than the base R functions.
- Remember that the following command

```
library(stringr)
```

# Aside: Two Common Errors

1. Need to *install* a package:
   - Observed error:

```
library(stringr)
```

```
Error in library(stringr) : there is no package called ‘stringr’
```

   - Fix:

```
install.packages("stringr")
```

2. Need to *load* a package:
   - Observed error:

```
str_detect("a", "apple")
```

```
Error in str_detect("a", "apple") : could not find function "str_detect"
```

   - Fix:

```
library(stringr)
```

# String Creation

- In R, we can create a string using single or double quotes.
- The convention is to use double quotes, and to use single quotes within a string if necessary.
- Here are some examples:

```
string1 <- "This is a string"
string2 <- "A 'string' within a string"
string3 <- c("one", "two", "three")

# Computes the length of each string:
str_length(string3)
```

```
[1] 3 3 5
```

# Alternative to paste()

- To combine strings, we can use `str_c` instead of `paste()`.

```
str_c("x", "y", "z")
```

```
[1] "xyz"
```

```
str_c("x", "y", "z", sep=',')
```

```
[1] "x,y,z"
```

```
str_c("x", c("a", "y"), "z", sep=',')
```

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

# Hawker IDs

- For instance, if we wished to create a sequence of hawker centre IDs for the centres earlier, we could do:

```
hawk_ids <- str_c("hawker", "ctre", 1:116, sep="_")
head(hawk_ids)
```

```
[1] "hawker_ctre_1" "hawker_ctre_2" "hawker_ctre_3"
[4] "hawker_ctre_4" "hawker_ctre_5" "hawker_ctre_6"
```

# Subsetting Strings

- To subset a string, we can use the function `str_sub()`.

```
x <- c("apple", "banana", "pear")
str_sub(x, 1, 3)
```

```
[1] "app" "ban" "pea"
```

```
str_sub(x, 1, 20)    # Useful when length is unknown
```

```
[1] "apple"  "banana" "pear"
```

# Regular Expressions

- What we will need most when parsing strings, are **regular expressions**.
- These are a sort of language for us to communicate what we are searching for to the computer.
- They are used in many programming languages, not just R, so it is worth knowing a little about them.

# Basic Matches

- The function `str_view()` lets us test out a regular expression.

```
# If I supply the expression as 'an',
# where will it match?

x <- c("apple", "banana", "pear")
str_view(x, "an")
```

# Basic Matches

cont'd

- To match an **a** at the beginning of a string,

```
str_view(x, "^a")
```

- To match an **a** at the end of a string,

```
str_view(x, "a$")
```

- To match an **a** or an **e** at the end of a string,

```
str_view(x, "[ae]$")
```

- To match a string of 3 characters with **a** in the middle,

```
str_view(x, ".a.")
```

# Detecting Matches

- To actually detect the match, we use `str_detect`

```
str_detect(x, "an")
```

```
 [1] FALSE  TRUE FALSE
```

```
str_detect(x, ".a.")
```

```
 [1] FALSE  TRUE  TRUE
```

# Hawker Centres in Ang Mo Kio

- Let us try to identify the hawker centres in Ang Mo Kio from the dataset that we have.

```
tf_vec <- str_detect(street_names, "Ang Mo Kio")
amk_id <- which(tf_vec)
amk_id
```

```
[1]  46  47  51  56  59  78  82  86 115
```

- Thus, in total, there were 9 hawker centres in Ang Mo Kio.
- Using `str_detect` followed by `which` is so common that there is a shortcut in the package:

```
str_which(street_names, "Ang Mo Kio")
```

```
[1]  46  47  51  56  59  78  82  86 115
```

# More Complex Regular Expressions

- Suppose we wish to extract all hawker centres in Jurong or Boon Lay.
- We could use the following code:

```
tf_vec <- str_detect(street_names, "Jurong|Boon Lay")
street_names[which(tf_vec)]
```

- Suppose we wish to extract all hawker centres that end with “Road”, and only one name preceding it:

```
tf_vec <- str_detect(street_names, "^(\\w)+ Road$")
street_names[which(tf_vec)]
```

# Character Classes

- The rectangular parentheses define character classes. They match as long as any one of the characters matches.

```
y <- paste("Ang Mo Kio", c(1:3,11))
str_detect(y, "Ave [13]$")
```

```
[1] TRUE FALSE  TRUE FALSE
```

- We can add modifiers after a character (or character class) to specify how many times a preceding character should match.

```
str_detect(y, "Ave [13]{2}$")    # match exactly twice
str_detect(y, "Ave [13]{1,2}$")  # match exactly once or twice
str_detect(y, "Ave [13]+$")      # match one or more times.
str_detect(y, "Ave [13]?$")      # match zero or one time.
```

# Character Classes

cont'd

- R provides several pre-defined character classes that we can use.
- For instance,
  - `[:alpha:]` matches any alphabetic character (lower or upper case)
  - `[:punct:]` matches any punctuation character.

```
sent1 <- "She said, \"I'd like 10 eggs, please.\""

str_view_all(sent1, "[:punct:]")
str_view_all(sent1, "[[:digit:][:punct:]]")
str_view_all(sent1, "[:space:]")
```

- Take a look at the help page `?base::regex` for more information on the available character classes within R.

# Groups

- We can use (normal) parenthesis to define groups, which can then be retrieved by their position.

```
sent_type <- "I went to the the shop."
str_view_all(sent_type, "\\b(\\w+)\\b \\1")

str_replace(sent_type, "\\b(\\w+)\\b \\1", "\\1")
```

```
[1] "I went to the shop."
```

- There can be more than one group in the pattern:

```
y <- paste("Ang Mo Kio", c("Ave", "St."), c(2, 64))
str_match(y, "(Ave|St.) ([0-9]+)$")
```

```
     [,1]     [,2]  [,3]
[1,] "Ave 2"  "Ave" "2"
[2,] "St. 64" "St." "64"
```

# Look-ahead and Look-behind Operators

- These operators are special, because they allow to extract strings according to what is behind or ahead of them.
- Look ahead:

```
str_extract(sent1, "[0-9]+(?= eggs)")
```

```
[1] "10"
```

- Look behind:

```
str_extract(sent1, "(?<=She said,).+")
```

```
[1] " \"I'd like 10 eggs, please.\""
```

# Overview of String Manipulations

1. `str_detect` is used to detect a pattern.
2. `str_replace` is used to replace a pattern.
3. `str_extract` is used to extract a pattern.
4. `str_match` is used to extract matching groups.
5. `str_split` is used to split a string according to a pattern.

# More Help on Regular Expressions in R

- The vignettes from the package provide good explanations:

```
vignette('stringr')                # introduction to the package
vignette('regular-expressions')    # covers regexes
```

- The following page offers even more details:

```
?about_search_regex
```

- The following page lists all the R-specific character classes defined:

```
?base::regex
```

1. Installing R and RStudio
2. R Objects
3. R Syntax
4. Conditional Execution and Loops
5. Functions
   - In-built R Functions
   - Writing R Functions
   - The `apply` Family of Functions (also in-built)
6. Important Classes of Objects
   - Strings
   - Factors
   - Date Objects
7. Basic Plotting With R
   - Line and Scatterplots
   - Barcharts
8. Generating Reports With R Markdown

# Introduction

- We use factors when we work with categorical variables in R.
- These are variables that have a fixed and known set of possible values.
  - Examples are disease status (2 values) or calendar month (12 possible values).
- We shall see that they are useful for dividing our into dataset into groups and performing analyses.
- This is a technique that you will use time and again to understand the patterns in your data.

# Creating Factors

- Suppose that we have a data vector containing month names:

```
x1 <- c("Dec", "Apr", "Jan", "Mar", "Apr")
```

- We can create a factor using the following command:

```
x2 <- factor(x1)
x2
```

```
 [1] Dec Apr Jan Mar Apr
Levels: Apr Dec Jan Mar
```

# Levels of a Factor

- The levels of a factor are the possible values that that variable could take on.

```
levels(x2)
```

```
[1] "Apr" "Dec" "Jan" "Mar"
```

- This does not seem right. R needs to be told about the remaining months, even though they do not appear in our dataset.

```
x3 <- factor(x1, levels=c("Feb", "Jan", "Mar", "Apr",
                          "May", "Jun", "Jul", "Aug",
                          "Sep", "Oct", "Nov", "Dec"))
x3
```

```
[1] Dec Apr Jan Mar Apr
Levels: Feb Jan Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

# Re-Ordering Levels

- If you noticed, `Feb` comes first when the levels are listed.
- When we print a graph, this means that `Feb` will come before `Jan`.
- We can correct this with the following code. Note that we have not created **ordered factors**, we have just changed which will be listed first.

```
x3 <- factor(x1, levels=c("Jan", "Feb", "Mar", "Apr",
                          "May", "Jun", "Jul", "Aug",
                          "Sep", "Oct", "Nov", "Dec"))
```

```
[1] Dec Apr Jan Mar Apr
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

1. Installing R and RStudio
2. R Objects
3. R Syntax
4. Conditional Execution and Loops
5. Functions
   - In-built R Functions
   - Writing R Functions
   - The `apply` Family of Functions (also in-built)
6. Important Classes of Objects
   - Strings
   - Factors
   - Date Objects
7. Basic Plotting With R
   - Line and Scatterplots
   - Barcharts
8. Generating Reports With R Markdown

# Date Class

- R contains a `Date` class to work easily with dates.
- Dates are stored internally as integers since 1st Jan 1970.
- This allows R to compute difference between dates, sequences of dates and divide dates into convenient periods.

# Creating Date Objects

- The easiest way is to create it from character strings:

```
d1 <- as.Date("2014/02/22", "%Y/%m/%d")
class(d1)
```

```
[1] "Date"
```

```
d1
```

```
[1] "2014-02-22"
```

# Convenience Functions

- There are several convenience functions for extracting information that we typically need from `Date` objects.

```
weekdays(d1, abbreviate=FALSE)
```

```
[1] "Friday"
```

```
months(d1, abbreviate=FALSE)
```

```
[1] "February"
```

- Later on, when we encounter the `tidyverse` environment, we shall introduce functions from the `lubridate` package that provides even more convenience functions.

# Sequence of Dates

- The `seq()` function works just as well with `Date` inputs.
- The following code creates a sequence of dates starting 100 days ago
- The values in the sequence are 7 days apart.

```
today <- Sys.Date()
s1 <- seq(today - 100, today, by="1 week")
s1[1:3]
```

```
[1] "2017-04-13" "2017-04-20" "2017-04-27"
```

# Dividing A Sequence of Dates into Groups

- At times, we need to group all dates that fall into a month, or week, or quarter together.
- We can do this easily with the `cut()` function.

```
cut(s1, breaks="month", labels=FALSE)
```

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

- There are analagous functions for time objects in R. We shall learn about them in due course.

1 Installing R and RStudio

2 R Objects

3 R Syntax

4 Conditional Execution and Loops

5 Functions
- In-built R Functions
- Writing R Functions
- The `apply` Family of Functions (also in-built)

6 Important Classes of Objects
- Strings
- Factors
- Date Objects

7 Basic Plotting With R
- Line and Scatterplots
- Barcharts

8 Generating Reports With R Markdown

# Creating a Scatterplot

- The default `plot()` function takes arguments `x` and `y`, which should be vectors of the same length, and produces a simple scatterplot.
- The arguments could also be
  - a list with components `x` and `y`, or
  - a data frame, or
  - a matrix with 2 columns.
- The axes, scales, titles and plotting symbols are all chosen automatically.
- However, these can be overridden.

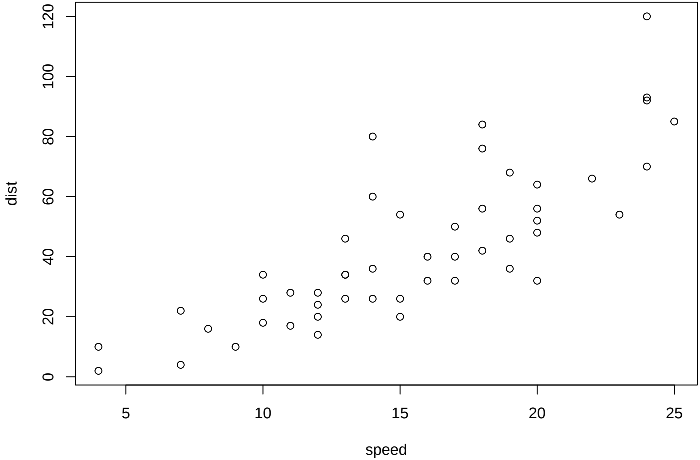
# Cars Dataset

- The `cars` dataset contains two columns:
  - speed in miles per hour.
  - distance to come to a stop, in feet.
- The data was collected in the 1920s, so it would not be applicable to today's cars.
- The following command creates a basic plot.

```
plot(cars)
```

# Cars Dataset

basic plot



# Cars Dataset

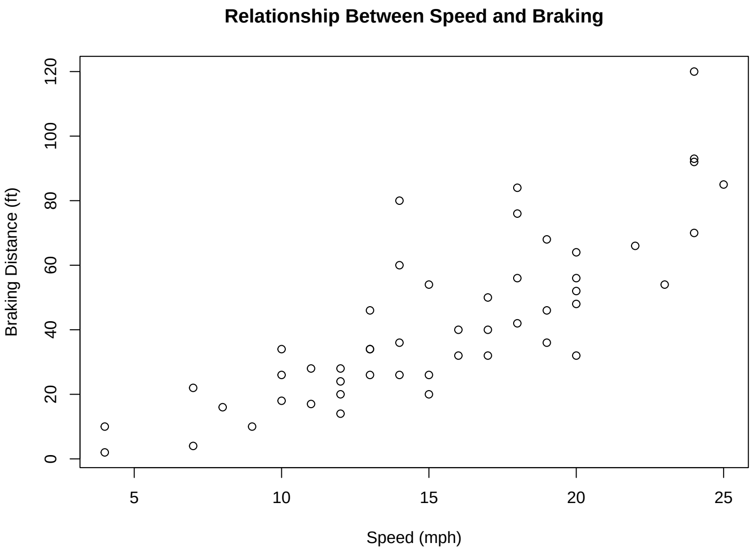
code for a plot with details

- By overriding some of the default arguments to R, we can create a more informative plot.

```
plot(cars, xlab="Speed (mph)",
     ylab="Braking Distance (ft)",
     main="Relationship Between Speed and Braking")
```

# Cars Dataset

a plot with details



# Altering Symbols

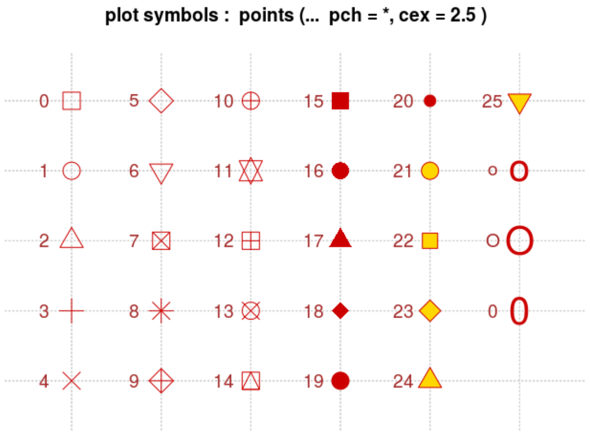
- In R, the plotting symbols are referred to as the *plotting character*.
  - They are referred to as `pch` for short.
- The actual symbol can be changed by specifying the `pch` argument to `plot()`.
- The full list of symbols is displayed on the next slide.
- For instance, the following command will plot (unfilled) triangles instead of unfilled circles.

```
plot(cars, pch=2)
```

- The default plotting character is `pch=1`

# Altering Symbols

plotting characters



# Altering Symbols

size of plotting characters

- To change the size of the plotting character, we need to modify the `cex` argument.
- This argument stands for "character expansion". The default value is 1.
- Larger values will make the symbol larger, and smaller values make it smaller.
- This is an important abbreviation, because you will see a similar argument in a lot of the help pages referring to other parameters:
  - `cex.axis` affects the font size in the axis.
  - `cex.main` affects the font size in the title, and so on.

# Altering Symbols

## adding colours

- The colour of the plotting characters can be changed using the `col` argument of the `plot()` function.
- The common colours can all be accessed by their names. For instance, the following command will create a plot with blue symbols.

```
plot(cars, col="blue")
```

- To see a list of all named colours in R, run the following command:

```
colours()
```

# Altering Symbols

cont'd

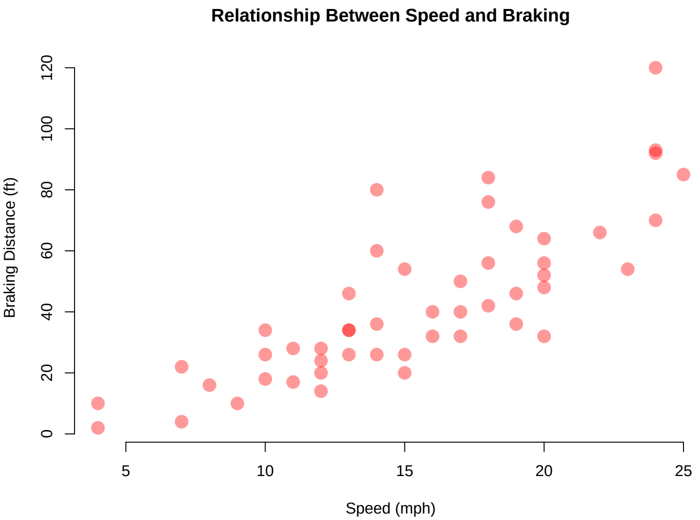
- When there are points that are very close to each other, it is useful to plot them using semi-transparent colours.
  - We shall see an alternative to this approach, jittering, much later in the class.
- To use this feature with base R plotting, we have to create the colour ourselves.

```
new_red <- rgb(1, 0, 0, alpha=0.4)
plot(cars, col=new_red, pch=19, cex=1.8, bty='n',
     xlab="Speed (mph)", ylab="Braking Distance (ft)",
     main="Relationship Between Speed and Braking")
```

- Read the help pages to understand what the new arguments do, or simply play around with them to see their effect.

# Altering Symbols

cont'd

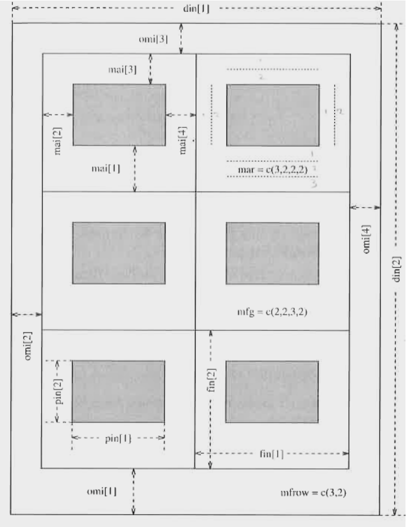


# Scatterplot Recap

- To add points and/or lines to an exising plot, use the `points()` and `lines()` functions. Using `plot()` again will re-draw the entire plot.
- The arguments pertaining to colour, size, shape, and labels all apply to other plots in base R as well.
- Do keep them in mind whenever you wish to fine-tune a default plot that R returns you.

# R Base Graphics Parameters

- On the right are some of the common parameters that are used to control base R graphics devices.
- We shall see a little more about this in the next chapter, but in general we shall stick to ggplot syntax for manipulating graphics in R.
- ggplot syntax is quite different from base R plotting.



1 Installing R and RStudio

2 R Objects

3 R Syntax

4 Conditional Execution and Loops

5 Functions
- In-built R Functions
- Writing R Functions
- The `apply` Family of Functions (also in-built)

6 Important Classes of Objects
- Strings
- Factors
- Date Objects

7 Basic Plotting With R
- Line and Scatterplots
- Barcharts

8 Generating Reports With R Markdown

# Barcharts

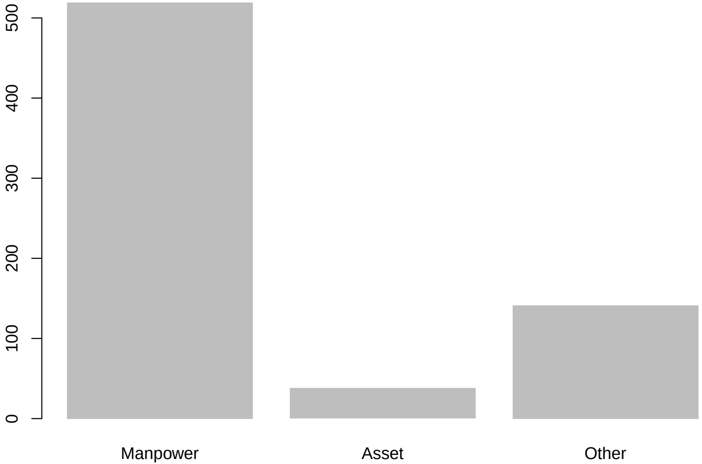
- Barcharts represent a variable by drawing bars whose heights are proportional to the values of the variable.
- We shall see a different interpretaion of the barchart when we cover the `ggplot2` package.
- For now, let us create a barchart using a data frame that we set up early on in this chapter.

| Expenditure category | Amount |
|---|---|
| Manpower | \$519.4m |
| Asset | \$38.0m |
| Other | \$141.4m |

# Barcharts

cont'd

```
barplot(op_budget$amount,  border=NA,
        names.arg = str_to_title(op_budget$exp_cat))
```



# Summary

- The base plotting commands in R are very powerful indeed.
- They give you full control over every single element in a plot window.
- For us though, we are going to focus on a slightly different paradigm when plotting – the grammar of graphics.
- We shall go into more detail about this later, but for now, this section was meant to give some knowledge on plotting with base R.
- As we proceed, we shall see more examples of plotting with base R until we reach the grammar of graphics topic.

1. Installing R and RStudio
2. R Objects
3. R Syntax
4. Conditional Execution and Loops
5. Functions
   - In-built R Functions
   - Writing R Functions
   - The `apply` Family of Functions (also in-built)
6. Important Classes of Objects
   - Strings
   - Factors
   - Date Objects
7. Basic Plotting With R
   - Line and Scatterplots
   - Barcharts
8. Generating Reports With R Markdown

# Introduction

- R Markdown is a language that allows you to combine code, its results and your text into one text document.
- That text document can then be "knitted" into a range of output formats, including html, pdf and Word.
- It is useful when you wish to
  - write a report based on your analysis (which is what you will be doing for the next few years in NUS).
  - share your work and findings with others. They will be able to easily reproduce your exact findings.
  - capture all your analyses on your dataset. This is useful, because everyone forgets!
- There is a new format that Rstudio is working on: Quarto. The latter is an engine that is agnostic to the language being used.
  - Read more about it here: `https://quarto.org/`
  - Don't worry though, R markdown is not going anywhere: `https://quarto.org/docs/faq/rmarkdown.html`

# R Markdown Help

As you begin, you will probably need to keep referring to these two documents:

- R Markdown Cheat Sheet: *Help* > *Cheatsheets* > *R Markdown Cheat Sheet*
- R Markdown Reference Guide: *Help* > *Cheatsheets* > *R Markdown Reference Guide*

# R Markdown Pre-requisites

- To generate HTML output, you do not need anything more than RStudio.
- To generate pdf output, you will need to have tex installed on your computer.
  - For windows, this means you will need MikTeX
  - For Mac OS, you will need to install MacTeX 2013+

# R Markdown Basics

- The first section of an Rmd file is usually a YAML header, surrounded by `---`s. It will look like this:

```
---
title: "Diamond sizes"
date: 2016-08-25
output: html_document
---
```

- YAML stands for Yet Another Markup Language. We will usually not have to write this ourselves. We specify certain options and RStudio will write this part for us.

# R Markdown Basics

cont'd

- The rest of the Rmd file will consist of code chunks (R code) and text.
- **Chunks** of R code will be surrounded by tickmarks:

````
```{r chunk_name}
Write R code here
```
````

- Text will be simple text, formatted with `#`, `*` and `_`.

# Chunk Names

Chunks can be given an optional name. This has three advantages:

1. You can more easily navigate to specific chunks using the drop-down code navigator in the bottom-left of the script editor.
2. Graphics produced by the chunks will have useful names that make them easier to use elsewhere.
3. You can set up networks of cached chunks to avoid re-performing expensive computations on every run.

# Chunk Options

1. `eval=FALSE` tells R to print the code, but not run it. This is useful for displaying example code.
2. `include=FALSE` tells R to run the code, but not include the output in the document.
3. `echo=FALSE` tells R not to show the code being run, but to include the output in the document.
4. `message=FALSE` and `warning=FALSE` suppress warning messages from appearing in your document.
5. `results='hide'` hides printed output, `fig.show = 'hide'` hides plots.

# Chunk Caching

- Sometimes, one or more of our code chunks is computationally expensive.
- In these cases, we do not want to run the chunk every time we knit the file.
- We would want to run it again only if some code in it has changed.
- In order to do this, we need to place the option `cache=TRUE` in the code chunk header.

# Learn More

- R Markdown is a great tool for sharing your work.
- Notice that you no longer have to zip up pdf output, source code and images to your colleagues or team-mates.
- Just one text file (Rmd), and they can do what you have done, exactly.
- It will take a short while to get used to the formatting. After that it will become very easy to use.
- I hope you find it useful once we leave this class!