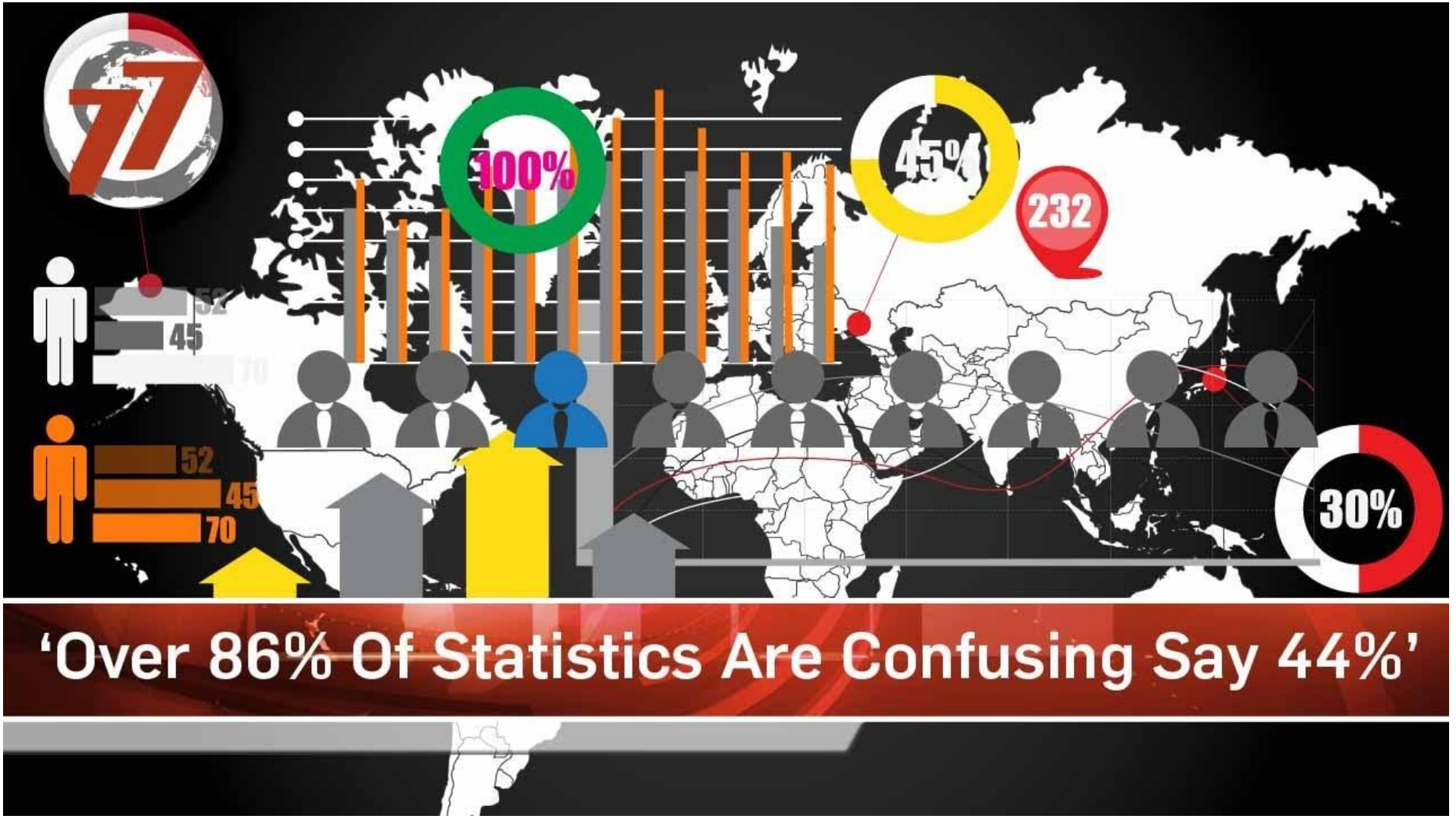
Turbocharge your statistics lessons using R!

Dr Tim Paulden

(Innovation & Development Manager, ATASS Sports)

CPD course (Mathematics Teachers' Network),
Exeter Mathematics School, 2 & 23 June 2015


77
100%
45%
232
52
45
70
52
45
70
30%
‘Over 86% Of Statistics Are Confusing Say 44%’


20

PREDICT THE PREM

Created by Dr Tim Paulden, ATASS Sports
tim.paulden@atass-sports.co.uk

Introduction

The goal of this project is for your team to build a mathematical model for predicting football results in the Premiership, and investigate different ways in which it might be extended and improved further.

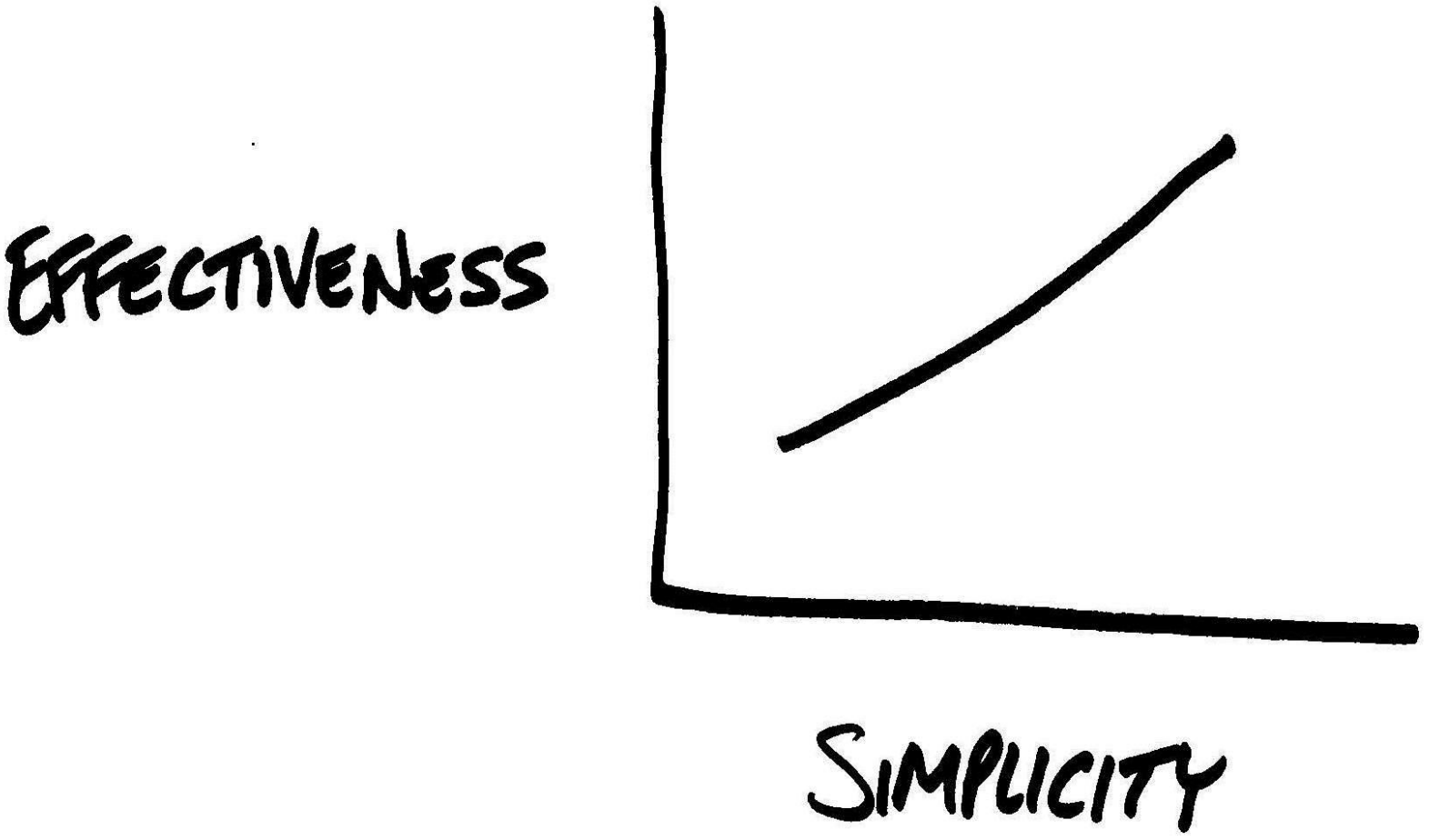
At the end of the project, you will generate score predictions for all the Premiership matches on the weekend after hand-in, using the best model that your team devises. Which team will produce the best predictions? Let's see!

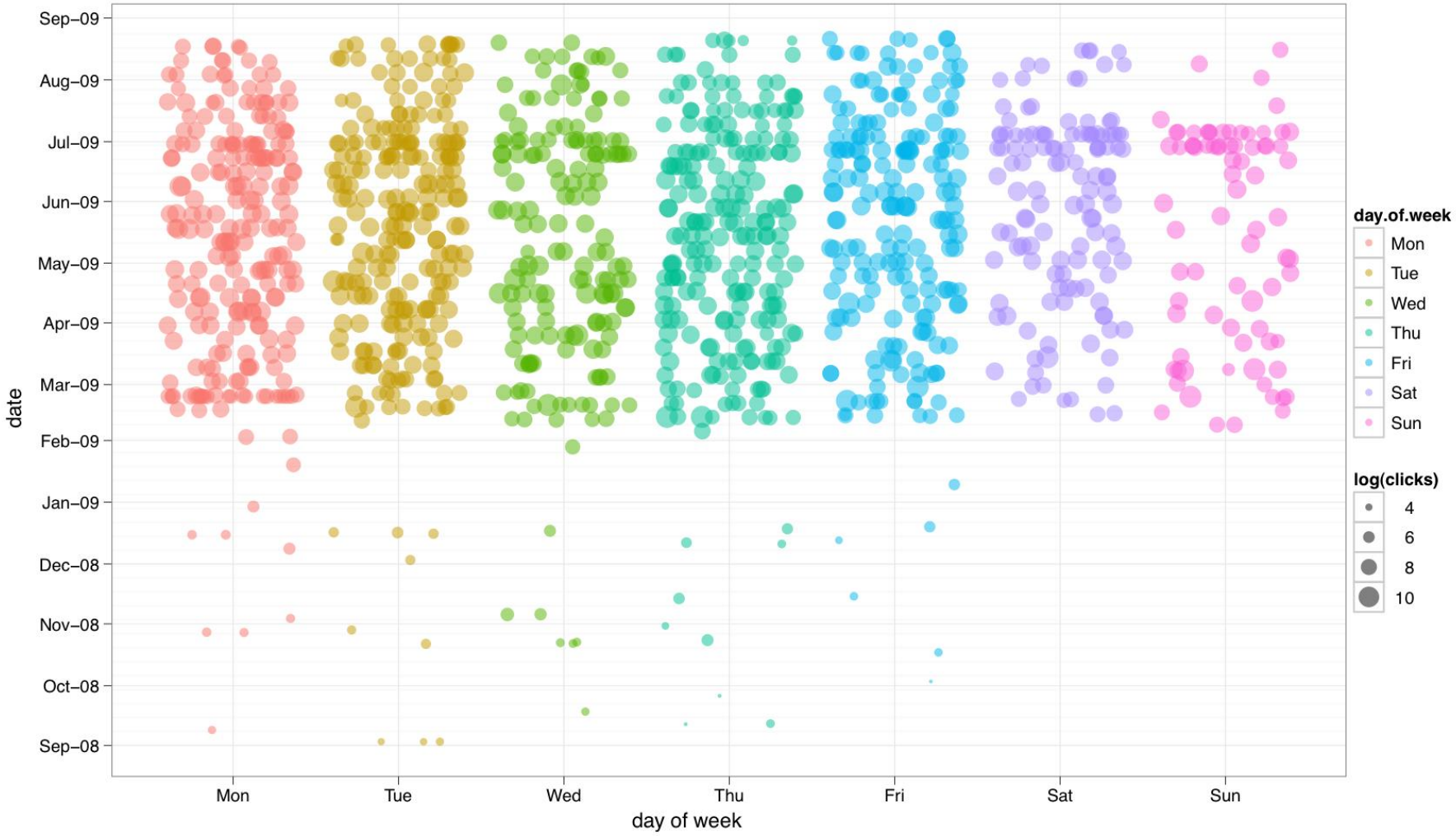
Stage 1

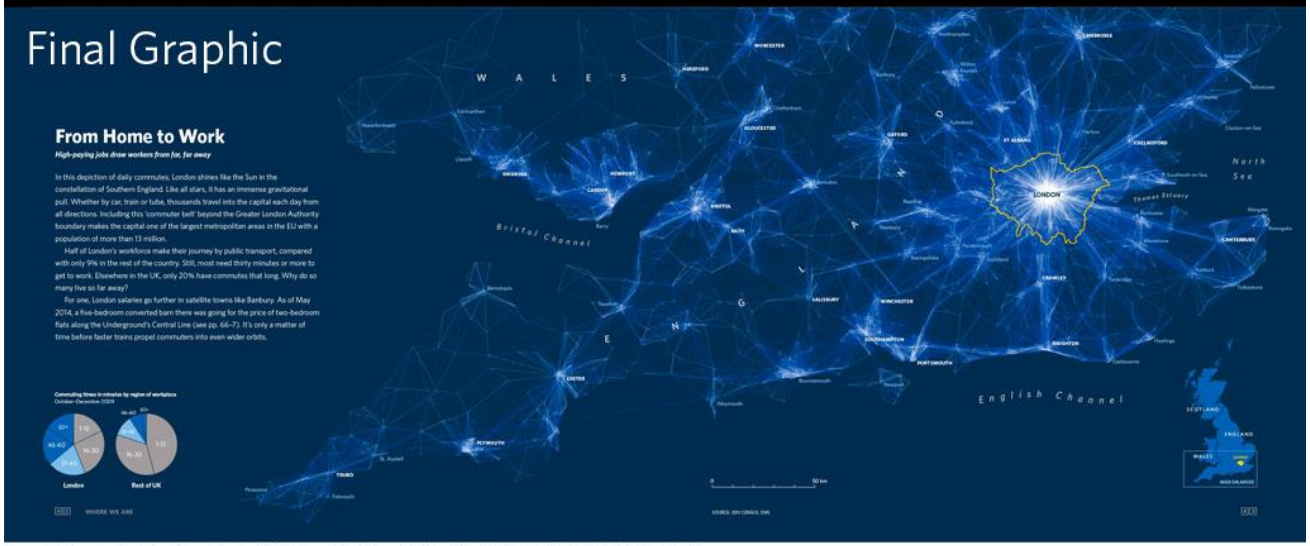
Begin by reading the 8 May 2010 news article "Predicting the Premiership" by Professor David Spiegelhalter (from Cambridge University), in which he generates score predictions for the ten matches on the last day of the 2009/10 season (http://news.bbc.co.uk/today/hi/today/newsid_8668000/8668153.stm).

Why R?

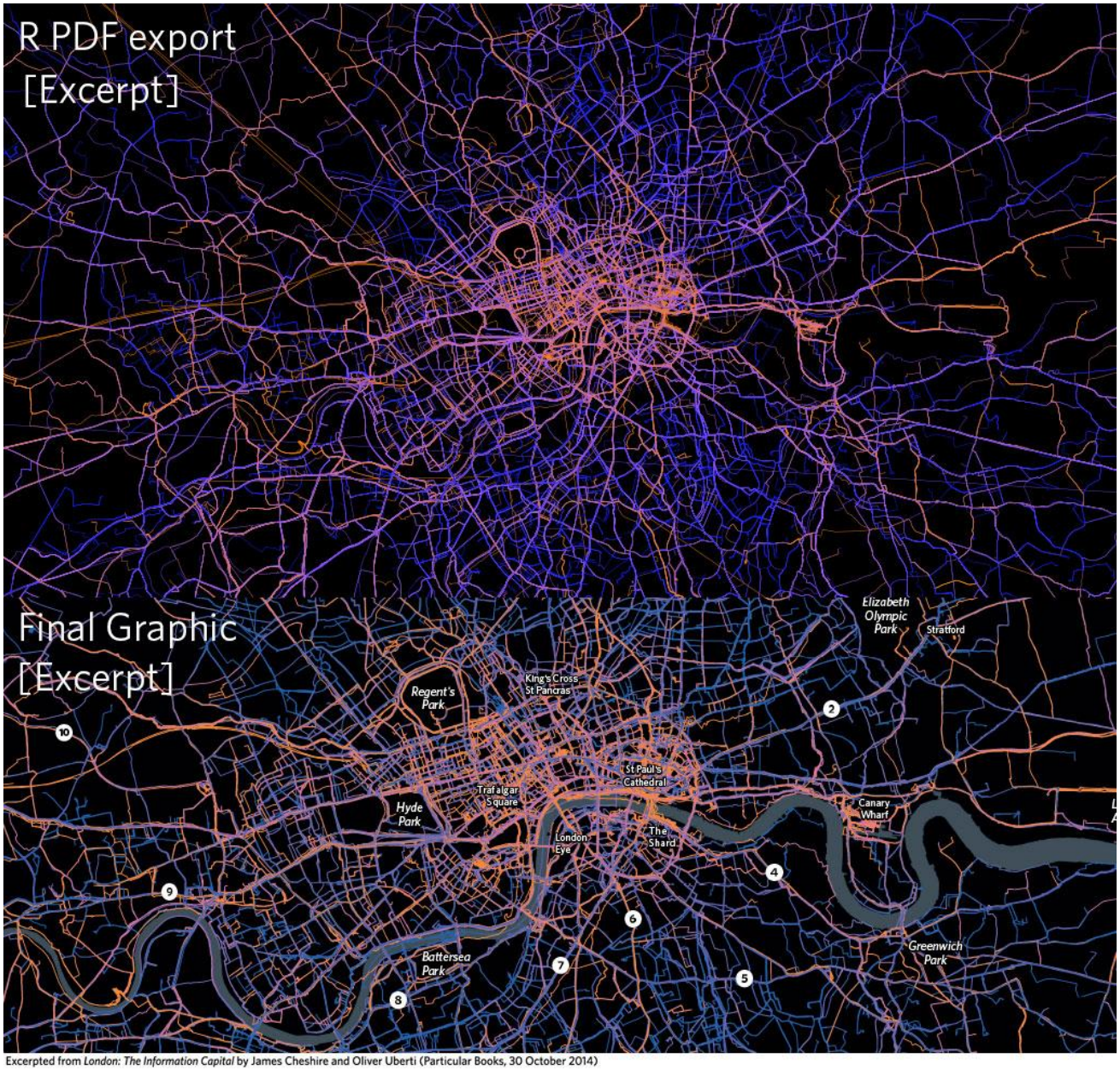

Free


EFFECTIVENESS
SIMPLICITY


Sep–09
Aug–09
Jul–09
Jun–09
May–09
Apr–09
Mar–09
Feb–09
Jan–09
Dec–08
Nov–08
Oct–08
Sep–08
date
Mon
Tue
Wed
Thu
Fri
Sat
Sun
day of week
day.of.week
Mon
Tue
Wed
Thu
Fri
Sat
Sun
log(clicks)
4
6
8
10



Excerpted from *London: The Information Capital* by James Cheshire and Oliver Uberti (Particular Books, 30 October 2014)



Excerpted from *London: The Information Capital* by James Cheshire and Oliver Uberti (Particular Books, 30 October 2014)

The FOUR V's of Big Data

From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Velocity, Variety and Veracity**

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States

Volume
SCALE OF DATA

40 ZETTABYTES
[43 TRILLION GIGABYTES]
of data will be created by 2020, an increase of 300 times from 2005

It's estimated that
2.5 QUINTILLION BYTES
[2.3 TRILLION GIGABYTES]
of data are created each day

6 BILLION PEOPLE
have cell phones

WORLD POPULATION: 7 BILLION

Most companies in the U.S. have at least
100 TERABYTES
[100,000 GIGABYTES]
of data stored

Variety
DIFFERENT FORMS OF DATA

As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
[161 BILLION GIGABYTES]

By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS

4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month

30 BILLION PIECES OF CONTENT
are shared on Facebook every month

400 MILLION TWEETS
are sent per day by about 200 million monthly active users

Velocity
ANALYSIS OF STREAMING DATA

The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session

Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure

By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth

Veracity
UNCERTAINTY OF DATA

1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions

in one survey were unsure of how much of their data was inaccurate

Poor data quality costs the US economy around
$3.1 TRILLION A YEAR

Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS

IBM


MU

April 02, 2014

Seven quick facts about R

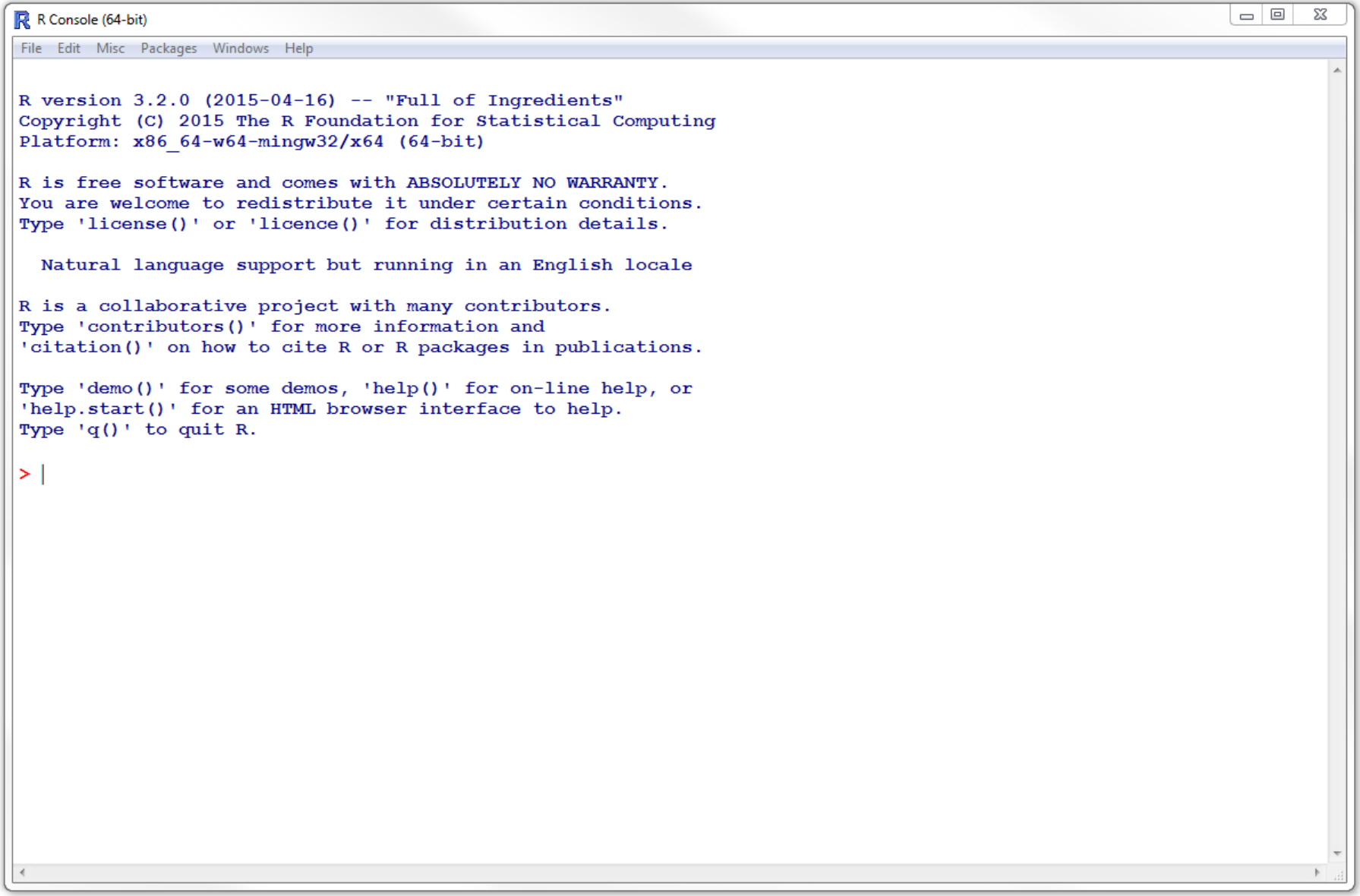
I've been spending the week at the Gartner Business Intelligence and Analytics Summit in Las Vegas, and R has been quite prominent here. Of course, R got namechecked several times on the panel about the Gartner Magic Quadrant for Advanced Analytics, and several of the regular talks mentioned R as well. I gave a short presentation on R and Revolution R Enterprise. You can see the slides below, and for Slide 4 I put together 7 facts about the growth of R:

1. R is the highest paid IT skill (Dice.com survey, January 2014)
2. R most-used data science language after SQL (O'Reilly survey, January 2014)
3. R is used by 70% of data miners (Rexer survey, October 2013)
4. R is #15 of all programming languages (RedMonk language rankings, January 2014)
5. R growing faster than any other data science language (KDNuggets survey, August 2013)
6. R is the #1 Google Search for Advanced Analytics software (Google Trends, March 2014)
7. R has more than 2 million users worldwide (Oracle estimate, February 2012)

For more analysis on the growth of R, see The Popularity of Data Analysis Software by Bob Muenchen, updated in March 2014 with new job trends statistics.

Posted by David Smith at 09:10 in popularity, R | Permalink

The basics of R


R Console (64-bit)
File Edit Misc Packages Windows Help
R version 3.2.0 (2015-04-16) -- "Full of Ingredients"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>

Using R like a calculator

- Start up "RGui" (a.k.a. the "R Console")
- Type the following into the white window:

```
1234 + 4567
[1] 5801
```

- What do you think the "[1]" means?
- How would you multiply 1234 and 4567?

Using R like a calculator

- Spaces within commands are ignored:

```
1234+4567
[1] 5801
```

- Use the up and down arrows to scroll through previously entered commands

Using R like a calculator

- Of course, R can also work with decimals:

```
22/7
[1] 3.142857
22/7 - pi
[1] 0.001264489
```

- By default, R shows you the answer to a small number of digits (7), but internally it stores numbers with much greater precision

Using R like a calculator

- The power symbol in R is ^ as shown below:

```
3^4
[1] 81
```

- Quick activity: Spend 20 seconds estimating the value of 9^6, then check using R

Using R like a calculator

- Of course, R follows BIDMAS:

```
10+20/5
[1] 14

20/5^2
[1] 0.8

(20/5)^2
[1] 16
```

Variables

- You can store the answer to a calculation in a variable (using whatever name you like):

```
myvar = 7*9
```

- To see the contents of a variable, simply type its name:

```
myvar
[1] 63
```

Variables

- And you can do any sort of operations you want on myvar – try it!

```
myvar/10
myvar^3
```

- You can call variables whatever you want, though it usually helps to use short names

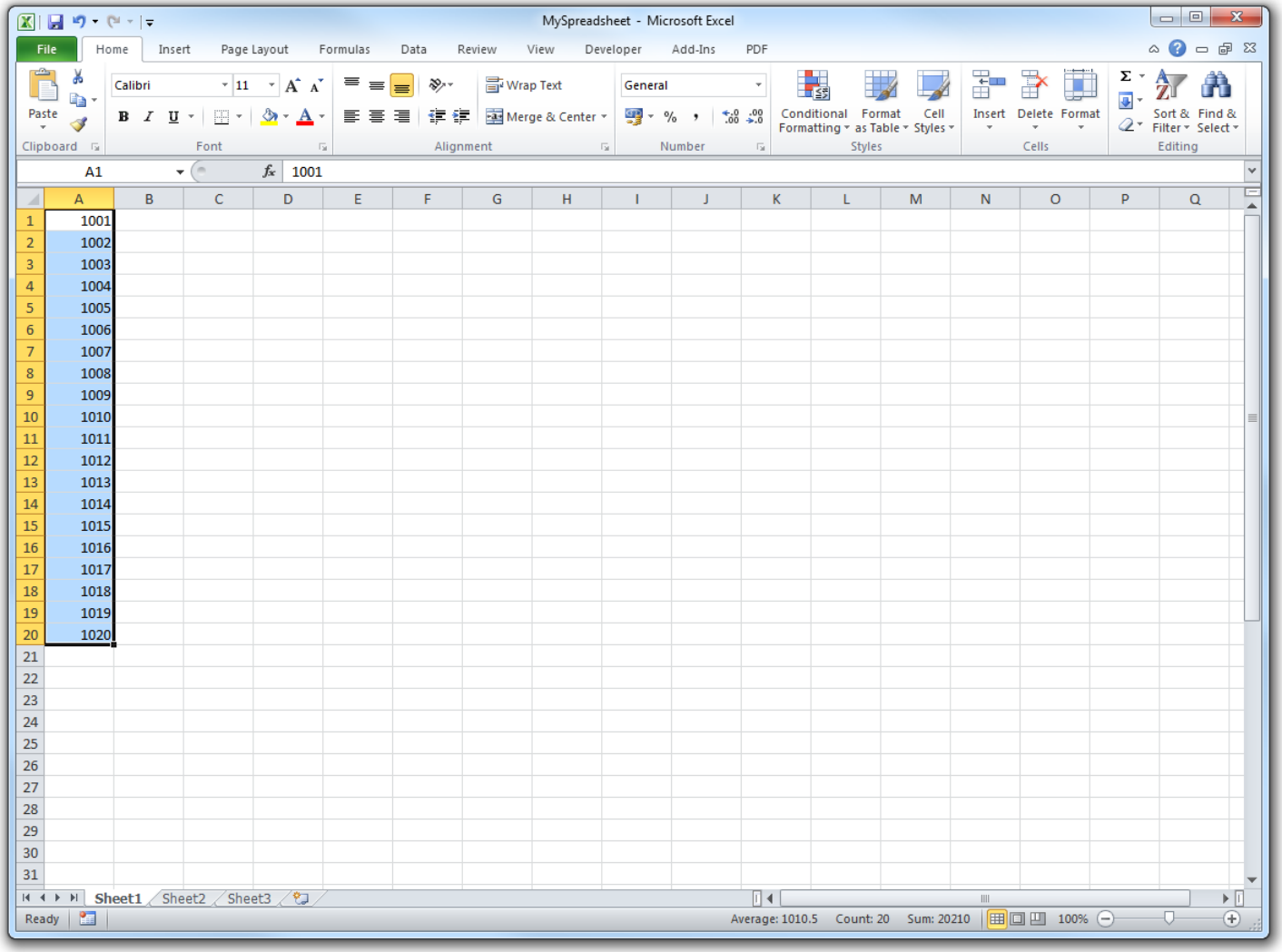
Introducing vectors

- Enter the following command, which gives the whole numbers from 1001 to 1020 inclusive:

```
1001:1020
```

- A list of numbers like this is known as a vector
- It might be useful to quickly draw an analogy between an R vector and a column of numbers in Excel…

Introducing vectors



Introducing vectors

- Now generate the numbers from 1001 to 2000 inclusive:

```
1001:2000
```

- The square brackets shown on the left of the resulting R output simply indicate the position in the vector (where the first position is 1, the second position is 2, and so on)

Introducing vectors

- The code below creates a vector called "myvec" containing the numbers from 11 to 70, and then displays the 17th number:

```
myvec = 11:70
myvec[17]
 [1] 27
```

- Quickly try a similar example for yourself

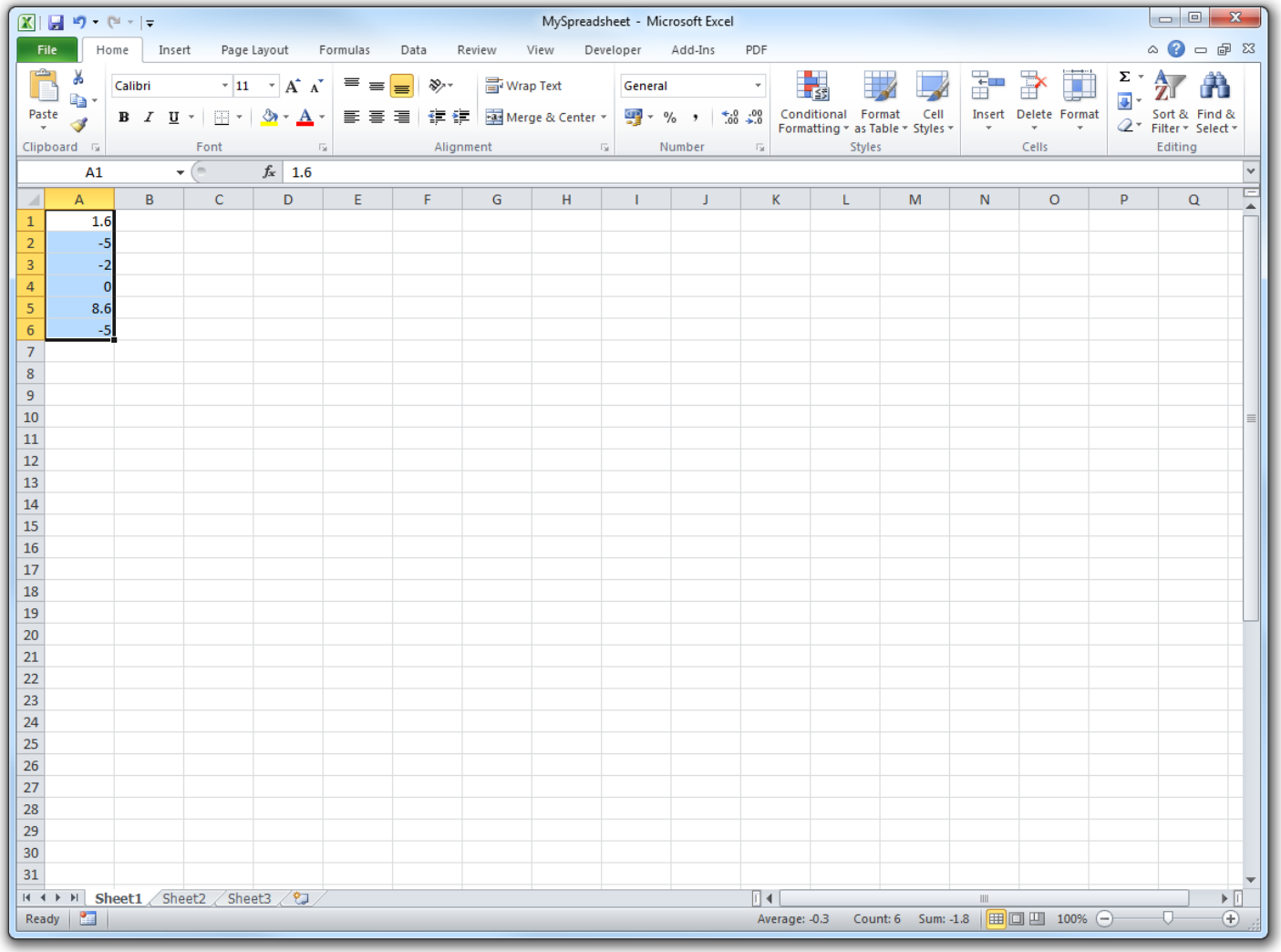
The combine command – "c"

- Another easy way to create a vector is using the "c" function (which stands for "combine"):

```
mynum = c(1.6, -5, -2, 0, 8.6, -5)
```

- As shown above, the numbers in a vector can be anything you want – including decimals, negatives, and zero – and can include repeats

The combine command – "c"



The combine command – "c"

- As before, we can refer to positions in the vector using square brackets:

```
mynum[3]
[1] -2
```

- Here's the clever bit… the square brackets can contain a vector of positions – for instance:

```
mynum[3:5]
[1] -2.0  0.0  8.6
```

The combine command – "c"

- Note that you can also use the "c" function to combine different vectors – for instance:

```
c(mynum, 10:20, mynum, -1, mynum)
```

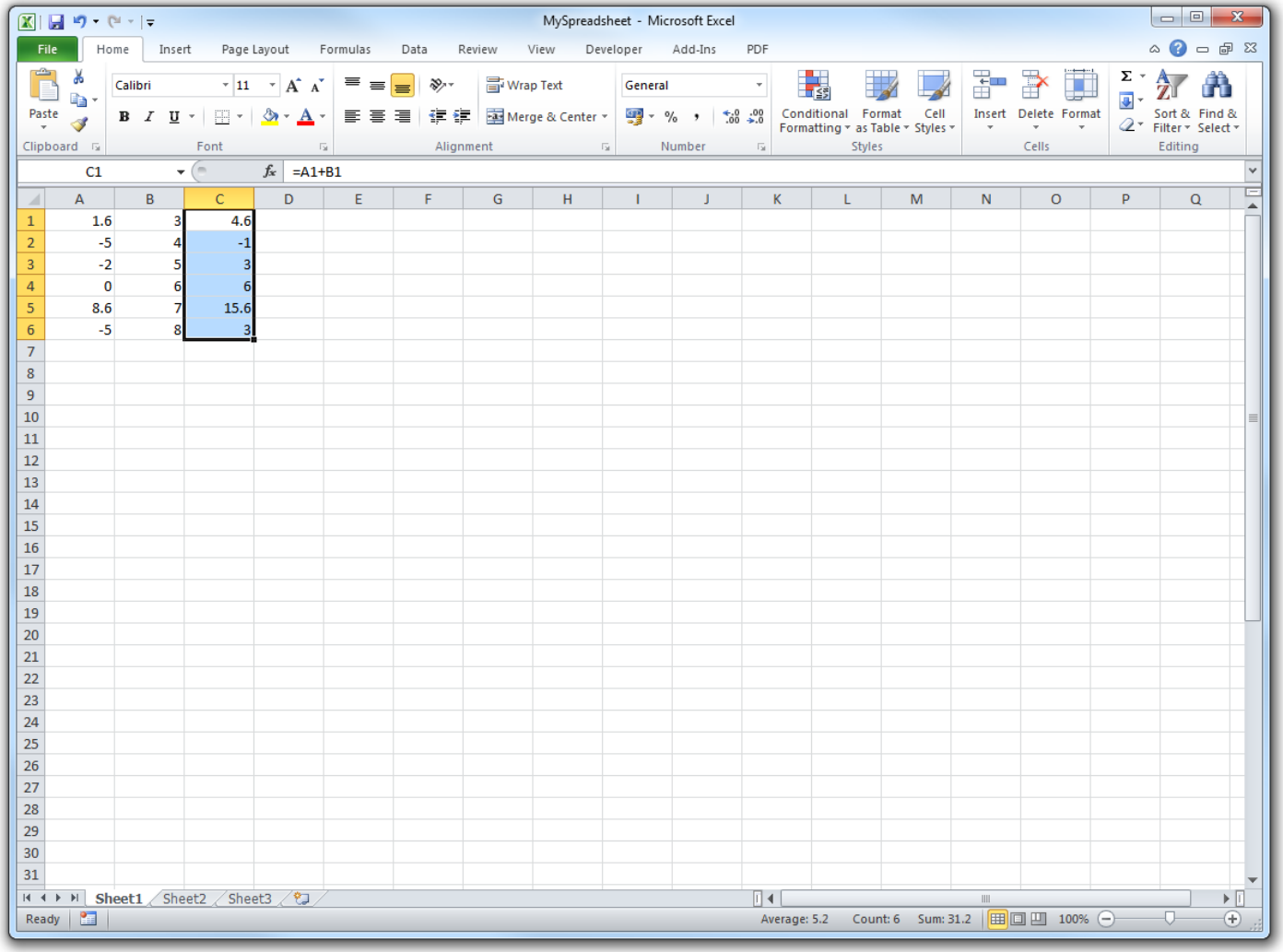
- Spend a few minutes trying it for yourself – create a few different vectors and then use the "c" command to combine them

Vector operations

- Vectors of the same length can be added, subtracted, multiplied, or divided – note that R does the calculation position by position:

```
vec1 = c(1.6, -5, -2, 0, 8.6, -5)
vec2 = c(3, 4, 5, 6, 7, 8)
vec1+vec2
vec1-vec2
vec1*vec2
vec1/vec2
```

Vector operations



	A	B	C
1	1.6	3	4.6
2	-5	4	-1
3	-2	5	3
4	0	6	6
5	8.6	7	15.6
6	-5	8	3

Vector operations

- R also knows how to arithmetically combine a vector with a single number

```
vec1 = c(1.6, -5, -2, 0, 8.6, -5)
vec1+3
vec1-3
vec1*3
vec1/3
```

A quick note on character vectors

- As we'll see later, R can also work with character vectors – i.e. vectors containing strings rather than numbers:

```
mych = c("exeter", "mathematics",
"school")
```

- The indexing notation is still exactly the same:

```
mych[c(2,3,1,3,2)]
```

Basic functions

- R has hundreds of built-in functions!

- To calculate the sum of the whole numbers from 1 to 100, just use the "sum" function:

```
sum(1:100)
[1] 5050
```

Basic functions

- Quick activity ideas:
  - How could you have figured out this answer in your head, without using R? (How many different "methods" can you think of?)
  - Using similar reasoning, can you predict the sum of the whole numbers from 1 to 1000? Use R to check your answer

```
sum(1:1000)
[1] 500500
```

Basic functions

- A few basic functions to be aware of:

```
mynum = c(1.6, -5, -2, 0, 8.6, -5)
length(mynum)
mean(mynum)
var(mynum)
sd(mynum)
min(mynum)
max(mynum)
range(mynum)
```

Basic functions

- Some more basic functions to be aware of:

```
mynum = c(1.6, -5, -2, 0, 8.6, -5)
summary(mynum)
quantile(mynum)
unique(mynum)
rev(mynum)
sort(mynum)
sort(mynum, decreasing=TRUE)
order(mynum)
```

Basic functions

- Note that functions in R always require you to use round brackets
- Elements in a vector are always selected using square brackets
- An example to illustrate:

```
sort(mynum[1:3])
```

Basic functions – which

- The "which" function is illustrated below:

```
which(mynum == -5)
[1] 2 6
which(mynum > 1)
[1] 1 5
```

- Caution: Don't put the < symbol immediately before a minus sign, as <- means "assign" in R

Basic functions – which

- It's common for the output from "which" to be used inside square brackets to filter down the data, based on a particular condition:

```
mynum[which(mynum>1)]
 [1] 1.6 8.6
mean(mynum[which(mynum>1)])
 [1] 5.1
```

Basic functions – table

- The table function can be particularly useful:

```
val = c(9, 7, 3, 7, 8, 3, 2)
table(val)
val
2 3 7 8 9
1 2 2 1 1
as.numeric(table(val))
[1] 1 2 2 1 1
```

Basic functions – table

- The table function can be particularly useful:

```
val = c(9, 7, 3, 7, 8, 3, 2)
table(val)
val
2 3 7 8 9
1 2 2 1 1
names(table(val))
[1] "2" "3" "7" "8" "9"
as.numeric(names(table(val)))
[1] 2 3 7 8 9
```

Basic functions – head/tail

- For long vectors, the functions "head" and "tail" are useful – these just show the start and end of the object (defaults to six values):

```
bigvec = 1001:10000
head(bigvec)
 [1] 1001 1002 1003 1004 1005 1006
tail(bigvec, 4)
[1]  9997  9998  9999 10000
```

Basic functions – rep

- The "rep" function can be used to repeat values – for instance:

```
rep(9, 10)
 [1] 9 9 9 9 9 9 9 9 9 9
rep(3:6, each=4)
 [1] 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6
rep(3:6, c(2,6,3,5))
 [1] 3 3 4 4 4 4 4 4 5 5 5 6 6 6 6 6
```

Basic functions – seq

- Enter the following command in R:

```
seq(7,30,4)
```

- What do you think the "seq" command does?

Basic functions – seq

- Quick activity idea involving sequences:
  - If a sequence starts with 169 and goes up by 987 each time, will it ever hit a power of ten?
  - What if the sequence starts with 169 and goes up by 985 each time?
- First part: yes – try small powers of ten:

```
tail(seq(169,1e6,987),3)
[1]  998026  999013 1000000
```

Extension question: will it hit another?

Basic functions – seq

- Quick activity idea involving sequences:
  - If a sequence starts with 169 and goes up by 987 each time, will it ever hit a power of ten?
  - What if the sequence starts with 169 and goes up by 985 each time?
- Second part: no – final digit is always 9 or 4, so never a power of ten (%% in R is "modulo")

```
myseq = seq(169,1e6,985)
myseq %% 10
```

Functions for character vectors

- It's worth noting that R also has a number of functions specifically for character vectors (we'll look at these in an activity later):

```
mych = c("exeter", "mathematics",
"school")
nchar(mych)
substr(mych, 1, 3)
grep("h",mych)
gsub("e","E",mych)
```

Flow commands: for, if/else, while

- The following code prints the message "Hello!" on the screen ten times:

```
for (jj in 1:10) {
  print("Hello!")
}
```

- This is an example of a "for loop". The block of code inside the braces is run ten times, with jj taking on each value from 1 to 10 in turn

Flow commands: for, if/else, while

- Let's change the code slightly to display the value of jj each time around:

```
for (jj in 1:10) {
  print(jj)
}
```

Flow commands: for, if/else, while

- We could also use the "cat" command to print a message containing the output:

```
for (jj in 1:10) {
  cat("Value of jj is", jj, "\n")
}
```

- The "\n" here simply indicates "new line"

Flow commands: for, if/else, while

- Let's now modify the code further to illustrate an "if" condition:

```
for (jj in 1:10) {
  if (jj>=7) {
    cat("Value of jj is", jj, "\n")
  }
}
```

Flow commands: for, if/else, while

- And here's an example of an if/else condition (note the position of the braces):

```
for (jj in 1:10) {
  if (jj>=7) {
    cat("Value of jj is", jj, "\n")
  } else {
    cat("jj is less than 7\n")
  }
}
```

Flow commands: for, if/else, while

- Of course, we could also use the value of jj in other ways – can you figure out the final value of "cursum" when the code below is run?

```
cursum = 0
for (jj in 1:20) {
  if (jj>cursum^2) {
    cursum = cursum + jj
  }
}
```

Flow commands: for, if/else, while

- Finally, here's a simple while loop:

```
curval = 240
while (curval>1) {
  cat("curval is", curval, "\n")
  curval = curval/2
  cat("new curval is", curval, "\n")
}
```

- The while condition (curval>1) is checked again each time the code in braces completes

Creating and running a script

- A series of commands can be stored as a text file with extension .r – known as an R script
- For instance, create a new file (in Notepad++) containing just the following command:

```
print("Hello Exeter!")
```

- Save the file somewhere on disk under the name **`myscript.r`** and then source it into R using the "source" function – for instance:

```
source("C:/scripts/myscript.r")
```

Creating and running a script

- You should see the message "Hello Exeter!" appear on the screen!
- Now extend your script by adding the six lines of code introduced a moment ago to illustrate a while loop (the code starting "curval = 240")
- Resave your script (as **`myscript.r`**) and source it again – does it run as expected?
- You can add comments to a script by including the character **#** before each comment

Some miscellaneous usage tips

- Seven quick tips:
  - Spaces in R commands are ignored
  - Use arrow keys to revisit/edit previous commands
  - To get help on a function, type the ? symbol followed by the function – e.g. `?var`
  - To see the objects in memory, type `ls()`
  - To remove an object called x, type `rm(x)`
  - To view command history, type `history(Inf)`
  - Avoid naming variables as: `c q t C D I T F`

Statistical examples

Dice rolling

- The following command selects a random number from the set {1, 2, 3, 4, 5, 6} (in other words, it represents a simulated dice roll):

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

- Run the command a few times to see that the output of the command varies

Dice rolling

- For rolling two dice, you might think the equivalent command would be

```
sample(1:6, 2)
```

but this isn't quite correct! Why not?

Dice rolling

- Answer: We need to sample with replacement, otherwise it's impossible to get the same number on both dice

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

- How could we confirm that this is doing the right thing?

Dice rolling – a quick simulation

- Let's roll the pair of dice repeatedly and find the total each time – we'll do 10000 repeats:

```
n = 10000
scores = rep(NA, n)
for (jj in 1:n) {
  roll = sample(1:6, 2, replace=TRUE)
  scores[jj] = sum(roll)
}
table(scores)
```

Dice rolling – creating a barplot

- We can use R to generate a simple bar chart to illustrate this:

```
barplot(table(scores))
```

- With labels and colour:

```
barplot(table(scores), xlab="Score",
ylab="Frequency", col="red")
```

Dice rolling – an animated version!

```
n = 10000
scores = rep(NA, n)
for (jj in 1:n) {
  roll = sample(1:6, 2, replace=TRUE)
  scores[jj] = sum(roll)
  if (jj%%50 == 0) {
    barplot(table(scores), xlab="Score",
    ylab="Frequency", col="red")
    Sys.sleep(0.1)
  }
}
```

Dice rolling – creating a histogram

- There is also a built-in histogram function, called "hist", though care should be taken with the breaks that are used, as these can significantly affect the appearance of the chart
- For instance, compare the following:

```
hist(scores)
hist(scores, breaks=seq(1.5,12.5,1))
```

Dice rolling – creating a histogram

- Strictly speaking, the y-axis in a histogram should represent frequency density (though if the bar widths are all equal then it makes no real difference to the chart's appearance)
- The following command shows the same histogram with density on the y-axis:

```
hist(scores, breaks=seq(1.5,12.5,1),
freq=FALSE)
```

Dice rolling – using set.seed

- To set the random seed in R, use "set.seed":

```
set.seed(1)
n = 10000
scores = rep(NA, n)
for (jj in 1:n) {
  roll = sample(1:6, 2, replace=TRUE)
  scores[jj] = sum(roll)
}
table(scores)
```

Uniform random numbers

- The runif command (which stands for "random uniform") can be used to generate random numbers from a uniform distribution

- For instance, to generate a single random number between 0 and 1, the command is:

```
runif(1,0,1)
```

Uniform random numbers

- Using R we can instantly generate a collection of random numbers from the uniform distribution – for instance:

```
mysample = runif(10000,0,1)
```

- If you take a large enough sample from the U[0,1] distribution, what values do the mean and variance approach? (Hint: The answer in each case is the reciprocal of a whole number)

Uniform random numbers

- Answer: Mean = 1/2, and variance = 1/12

Key Point

The Uniform random variable X whose density function $f(x)$ is defined by

$$f(x) = \begin{cases} \dfrac{1}{b-a}, & a \le x \le b \\ 0 & \text{otherwise} \end{cases}$$

has expectation and variance given by the formulae

$$E(X) = \frac{b+a}{2} \qquad \text{and} \qquad V(X) = \frac{(b-a)^2}{12}$$

Uniform random numbers

- Once again, we can use the hist function to get a graphical sense of what's going on:

```
mysample = runif(10000,0,1)
hist(mysample)
```

Activity idea: Higher or lower?

- Suppose we use runif to generate random numbers between 0 and 100 one at a time
- After each number is generated, we must guess whether the next number will be higher or lower than the current one
- Play the game for a few minutes in R. What do you think is the best strategy?

Activity idea: Higher or lower?

- Challenge: Using R, write a piece of code that follows the "optimal" strategy for the game – in other words, it simulates the game and automatically decides on each round whether to go "higher" or "lower"
- By simulating the game a large number of times (say 10,000 or 100,000), can you estimate the probability that the game lasts for N rounds (N = 1, 2, 3, …)?

How about other distributions?

- R can handle any sort of statistical calculations involving common distributions (including Bernoulli, Binomial, Poisson, Normal, …) including determining percentiles, generating random numbers, and producing plots
- Here are a few examples…

Example 1

- Plot showing proportion of heads obtained when flipping a fair coin:

```
nflip = 1000
ht = sample(0:1, nflip, rep=TRUE)
prop = cumsum(ht)/(1:nflip)
plot(1:nflip, prop, ylim=c(0,1),
type="l", xlab="Flip number",
ylab="Proportion of heads")
abline(h=0.5, col="red")
```

Example 2

- Charts showing Poisson distributions with different means:

```
barplot(dpois(0:30,3), names.arg=0:30,
xlab ="Value", ylab="Density")
barplot(dpois(0:30,15), names.arg=0:30,
xlab ="Value", ylab="Density")
```

Example 3

- Normal distribution plots for different means and variances:

```
xx = seq(-5,5,0.001)
plot(xx,dnorm(xx,0,1),col="red",
pch=".", xlab ="Value", ylab="Density")
points(xx,dnorm(xx,0,2), col="blue",
pch=".")
points(xx,dnorm(xx,2,1), col="green",
pch=".")
```

Example 4

- Histograms of random "height" data generated from a Normal distribution:

```
set.seed(1)
height10 = rnorm(10,175,8)
height100 = rnorm(100,175,8)
height1000 = rnorm(1000,175,8)
hist(height10, breaks=seq(135,215,1))
hist(height100, breaks=seq(135,215,1))
hist(height1000, breaks=seq(135,215,1))
```

A note on sample size

- For small sample sizes, data points generated from a distribution might be only weakly representative of the shape of the true distribution (such as the 10 draws from a normal distribution shown above)
- Using R, it is as easy to work with large data sets as small data sets!
- It is also easy to simulate data from scratch to explore statistical ideas

Linear regression with simulated data

- First, let's generate some simulated data…

```
set.seed(42)
x = rnorm(1000, 10, 5)
y = 3*x - 11 + rnorm(1000, 0, 5)

hist(x, breaks=50)
plot(x, y)
```

Linear regression with simulated data

- Fit a line of best fit (a "linear model" in R)

```
lm1 = lm(y ~ x) # standard linear model
summary(lm1)

# add line to the plot
plot(x, y)
abline(lm1, col="red")
abline(h=0)
abline(v=0)
```

Linear regression with simulated data

- Final example – an 'outlier':

```
x[17] = 4
y[17] = 60
lm1b = lm(y ~ x)
plot(x, y)
abline(lm1, col="red")
abline(lm1b, col="green")
abline(h=0)
abline(v=0)
```

A few words on data frames

Data frames

- Analogous to a whole sheet in Excel!
- The prem data set can be loaded into R using read.csv:

```
prem = read.csv("C:/RWorkshop/prem.csv",
as.is=TRUE)
```

- To see the data's dimensions:

```
dim(prem)
[1] 3800    7
```

Data frames

- Use the dollar notation (prem$...) to refer to a particular column (each of which is a vector):

```
sum(prem$hsc)
[1] 5846
```

- To see the column names:

```
colnames(prem)
[1] "date" "dayno" "seasnum" "ht" "at"
"hsc" "asc"
```

Data frames

- New columns can also be defined via the dollar notation:

```
prem$totsc = prem$hsc + prem$asc
head(prem)
```

- To view particular rows, use square brackets with a comma afterwards – for instance:

```
prem[1:3,]
```

Data frames

- Data can also be loaded in directly from csv files stored online, such as:

```
dat = read.csv("http://www.football-data.co.uk/mmz4281/1415/E0.csv")
```

- It's worth noting that there are a variety of data sets packaged with R – for details, type:

```
library(help = "datasets")
```

Packaging code as functions

Writing your own functions

- We'll start with a simple example – copy/paste the definition below into R:

```
CalcSquare = function(x) {
  y = x^2
  return(y)
}
```

- Now test the resulting function:

```
CalcSquare(7)
[1] 49
```

Writing your own functions

- Of course, you can also call the function and store the answer in a variable:

```
ans = CalcSquare(9)
ans
[1] 81
```

Writing your own functions

- If we pass a vector into the function, it will be squared:

```
CalcSquare(7:10)
[1] 49  64  81  100
```

Writing your own functions

- Here's an example for higher powers:

```
CalcPower = function(x, pow) {
  y = x^pow
  return(y)
}

CalcPower(7,3)
[1] 343
```

- How would you make pow default to 2?

Writing your own functions

- Modified version, including a default for pow:

```
CalcPower = function(x, pow=2) {
  y = x^pow
  return(y)
}

CalcPower(7,3)
[1] 343

CalcPower(7)
[1] 49
```

Writing your own functions

- R matches arguments "intelligently":

```
CalcPower = function(x, pow=2) {
  y = x^pow
  return(y)
}

CalcPower(po=3, x=7)
[1] 343

CalcPower(p=3, 7)
[1] 343
```

Writing your own functions

- Variables defined inside a function are local to that function!

```
CalcPower = function(x, pow=2) {
  y = x^pow
  return(y)
}

y = 11
CalcPower(7,3)
[1] 343
y
[1] 11
```

Writing your own functions

- Pass everything you need into the function – don't rely on variables “outside”, like this!

```
DodgyCalcPower = function(x) {
  y = x^pow
  return(y)
}

pow = 3
DodgyCalcPower(7)
[1] 343
```

Writing your own functions

- Finally, a list can be used to return more than one quantity from a function:

```
SumAndProduct = function(x, y) {
  return(list(ss = x+y, pp = x*y))
}

SumAndProduct(1:4, 7:10)
$ss
[1]  8 10 12 14

$pp
[1]  7 16 27 40
```

Writing your own functions

- Finally, a list can be used to return more than one quantity from a function:

```
SumAndProduct = function(x, y) {
  return(list(ss = x+y, pp = x*y))
}

SumAndProduct(1:4, 7:10)
$ss
[1]  8 10 12 14

$pp
[1]  7 16 27 40
```

Writing your own functions

- Final example: lm based on "simulated" data (as considered earlier):

```
SimLm = function(n=1000, a=3, b=-11, noisesd=5) {
  x = rnorm(n, 10, 5)
  y = a*x + b + rnorm(n, 0, noisesd)
  lm1 = lm(y ~ x)
  summary(lm1)
  plot(x, y)
  abline(lm1, col="red")
  abline(h=0)
  abline(v=0)
}
```

Writing your own functions

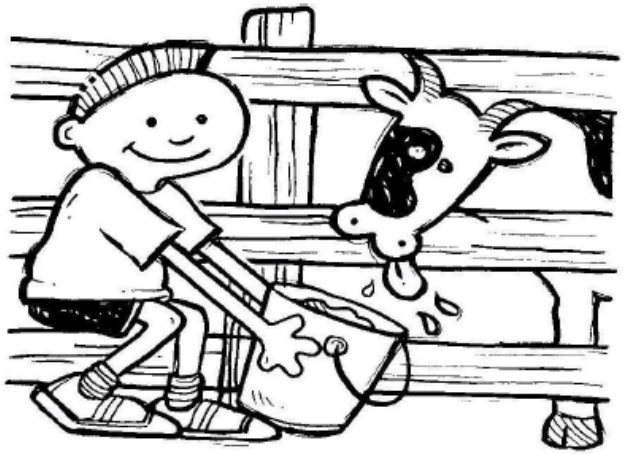
- Some thoughts on code infrastructure
  - If a collection of functions is available within a script, these functions can all be brought into R with one line of code (using "source")
  - The user can play with these functions without ever needing to look at the code inside them
  - Further down the road, it may be useful to include code to check inputs (using commands such as "stop"), but this is not yet a priority

Activity examples

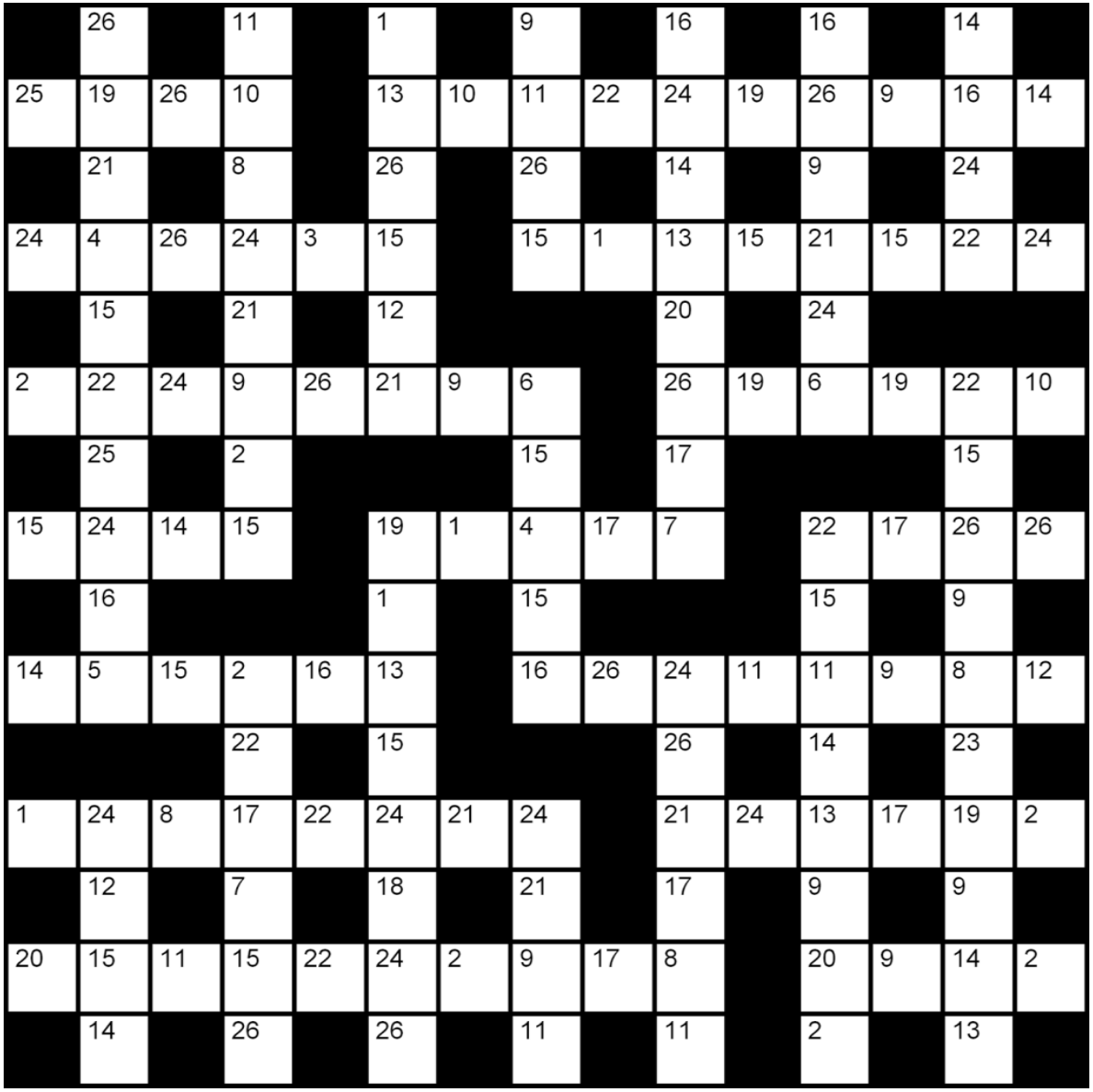
Activity:
Crack the Code!

Crack The Code

a=1	f=6	k=11	p=16	v=22
b=2	g=7	l=12	q=17	w=23
c=3	h=8	m=13	r=18	x=24
d=4	i=9	n=14	t=20	y=25
e=5	j=10	o=15	u=21	z=26



9			12	15	22		5		
13		25			16		5		2o


26 11 1 9 16 16 14
25 19 26 10 13 10 11 22 24 19 26 9 16 14
21 8 26 26 14 9 24
24 4 26 24 3 15 15 1 13 15 21 15 22 24
15 21 12 20 24
2 22 24 9 26 21 9 6 26 19 6 19 22 10
25 2 15 17 15
15 24 14 15 19 1 4 17 7 22 17 26 26
16 1 15 15 9
14 5 15 2 16 13 16 26 24 11 11 9 8 12
22 15 26 14 23
1 24 8 17 22 24 21 24 21 24 13 17 19 2
12 7 18 21 17 9 9
20 15 11 15 22 24 2 9 17 8 20 9 14 2
14 26 26 11 11 2 13

Text cleaning – a simple example

```
txt = c("  Hello! I'm a short paragraph (containing about ",
"50 words) that Tim has written to help illustrate a few ",
"text-processing functions in R.  As you can see, I'm split ",
"over a few lines--five, in fact--with each line consisting ",
"of several words. Hope it's all fairly self-explanatory! ")

txt = gsub("'", "", txt)
txt = gsub("-", " ", txt)
txt = tolower(txt)

txt = gsub("[^a-z ]", "", txt)

txt = gsub("^[ ]+", "", txt)
txt = gsub("[ ]+$", "", txt)
txt = gsub("[ ]+", " ", txt)
```

Isolating the words

```
wd = unlist(strsplit(txt," "))
```

Shuffling the letters

```
wd = unlist(strsplit(txt," "))

set.seed(123)
shuffle = sample(LETTERS)

cc = wd
for (i in 1:26) {
  cc = gsub(letters[i], shuffle[i], cc)
}
```

Activity: Devious Dice

- The “Devious Dice” on your tables – coloured Green, Red, and Blue – have unusual patterns of numbers on them… take a look
- We’re going to think about which dice is the best for playing “highest roll wins” head-to-head against another dice, and perform a simulation in R to confirm our answer
- The task would be really easy if the numbers on the three dice were something like 1,1,1,2,2,2 versus 3,3,3,4,4,4 versus 5,5,5,6,6,6… but for the Devious Dice, it’s less clear which is strongest

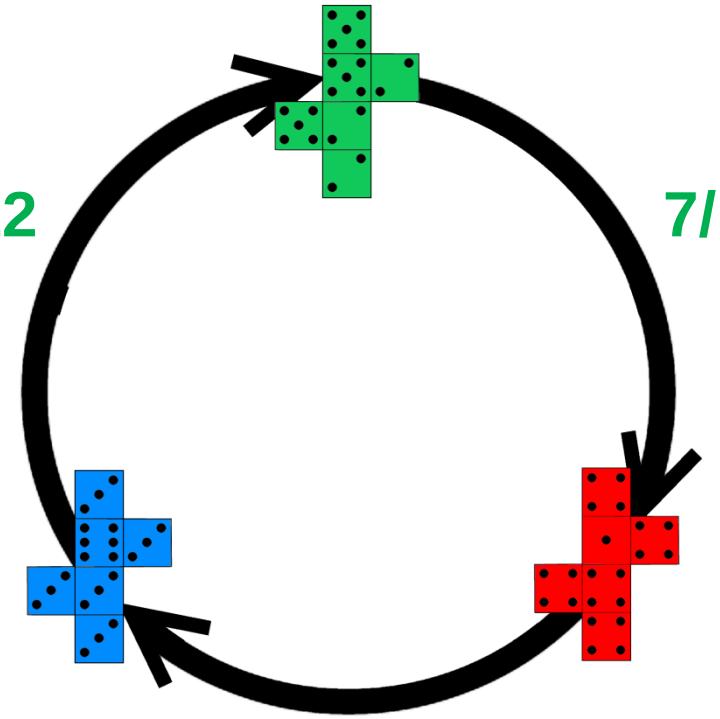
Devious Dice

- Pupils could spend a while experimenting with rolling these dice (e.g. play "first to win twenty rolls" in pairs) and see what patterns emerge
- What is the probability that Red beats Blue on a single roll? What about Blue beating Green on a single roll? Green beating Red on a single roll?
- Discuss in your group the simplest / most elegant method to generate these probabilities
- Which dice is the strongest?

Devious Dice

The three dice are designed to be "non-transitive"

7/12 vs 5/12



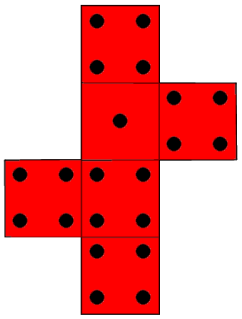
7/12 vs 5/12

25/36 vs 11/36

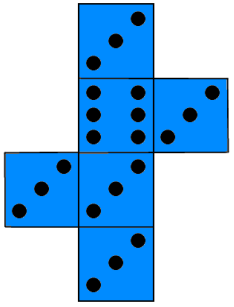
Devious Dice

- Counterintuitively, there is no best dice! This is tricky to explain, but the key idea is that “size of victory” is irrelevant – all wins count the same
- Observe that while each of the dice has 21 dots, when Red fights Blue, all Red's wins are always by just 1 dot (Red 4 vs Blue 3), while Blue's wins are always larger (either 2 or 5 dots)
- So Red wins more often, but by a smaller margin

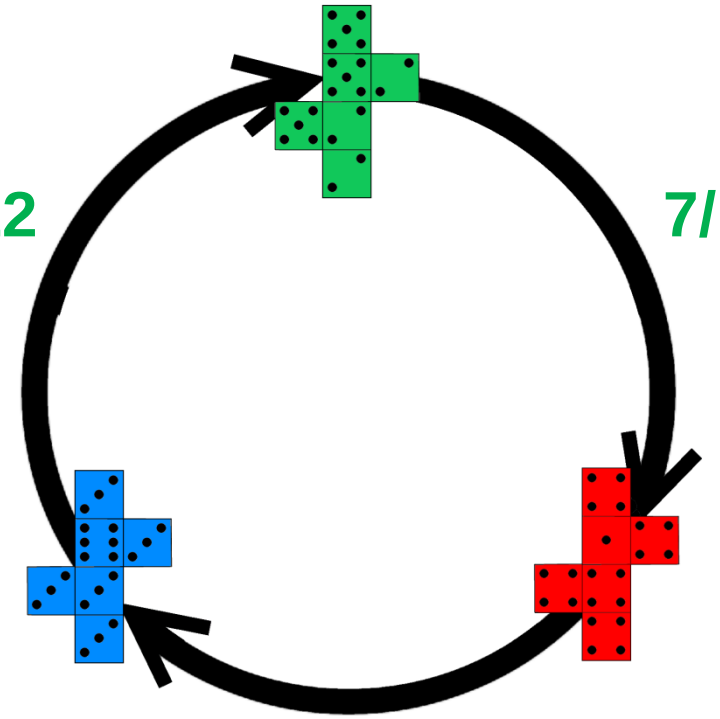
Devious Dice



25/36 **vs** **11/36**



This principle holds for each head-to-head…



7/12 vs 5/12

7/12 vs 5/12

25/36 vs 11/36

Devious Dice

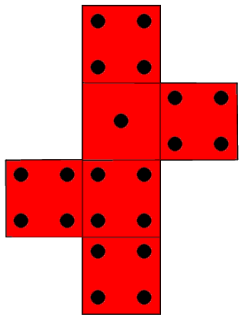
- We'll now illustrate how R can be used to generate these same answers via simulation
- Fundamentally, probability can be thought of as a description of what would happen if you were to look at a very large number of repetitions
- Probability in the real-world applications almost always involves computer simulations – not working by hand!

Devious Dice

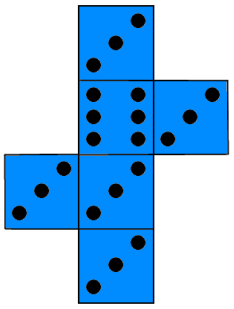
- Finding probabilities via computer simulation

```
set.seed(42)
n = 1000000
red = sample(c(1,4,4,4,4,4), n, replace=TRUE)
blue = sample(c(3,3,3,3,3,6), n, replace=TRUE)
table(red,blue)/n
    blue
red         3         6
  1 0.138791 0.027883
  4 0.693786 0.139540
36*table(red,blue)/n
    blue
red          3          6
  1   4.996476   1.003788
  4  24.976296   5.023440
```

Devious Dice



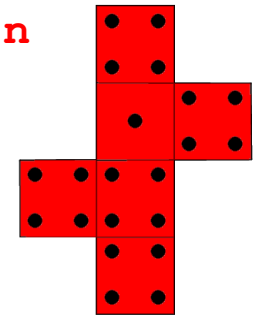
25/36 vs 11/36



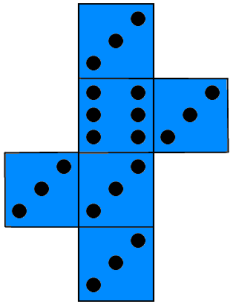
- Finding probabilities via computer simulation

```
set.seed(42)
n = 1000000
red = sample(c(1,4,4,4,4,4), n, replace=TRUE)
blue = sample(c(3,3,3,3,3,6), n, replace=TRUE)
table(red>blue)
 FALSE   TRUE
306214 693786
36*table(red>blue)/n
  FALSE    TRUE
11.0237 24.9763
36*length(which(red>blue))/n
[1] 24.9763
36*length(which(blue>red))/n
[1] 11.0237
```

Devious Dice



25/36 vs **11/36**



Something to ponder over lunch…

- If all three dice are rolled at the same time, which one is most likely to win?

Red: **1, 4, 4, 4, 4, 4**

Green: **2, 2, 2, 5, 5, 5**

Blue: **3, 3, 3, 3, 3, 6**

Devious Dice

- Finding probabilities via computer simulation

```
set.seed(42)
n = 1e7
red = sample(c(1,4,4,4,4,4), n, replace=TRUE)
green = sample(c(2,2,2,5,5,5), n, replace=TRUE)
blue = sample(c(3,3,3,3,3,6), n, replace=TRUE)

216*length(which(red>green & red>blue))/n
[1] 74.98816
216*length(which(green>red & green>blue))/n
[1] 90.00545
216*length(which(blue>red & blue>green))/n
[1] 51.00639
```

75/216 vs 90/216 vs 51/216

Devious Dice

- Can we reason these values out directly?

Red: 1, 4, 4, 4, 4, 4

Green: 2, 2, 2, 5, 5, 5

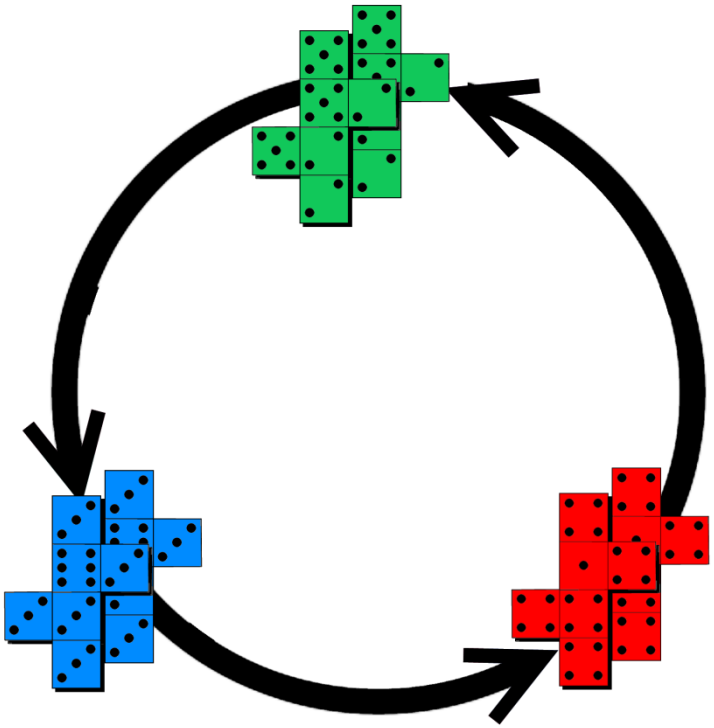
Blue: 3, 3, 3, 3, 3, 6

75/216 vs 90/216 vs 51/216

- The final surprise from the Devious Dice comes when we try rolling two dice of each colour rather than one (e.g. two Red versus two Blue), with each person adding their two rolls together
- What would you expect to happen?
- (A nice side point to make: a draw still isn't possible when two dice are used – why?)

Devious Dice

Amazingly, the cycle reverses!



TWO DICE

Devious Dice

Over to you…!

Thanks for your attention!

Dr Tim Paulden

tim.paulden@atass-sports.co.uk