Introduction to R
a brief overview

Kevin Keenan
kkeenan02@qub.ac.uk

QUB

September 16, 2013

"Education is not the filling of a pale, but the lighting of a fire."

W.B. Yeats
(1865-1939)

Session structure

- Lecture: 1hr
  - What is R
  - What can you do in R
  - R as a programming language
  - R as a statistical package
- Computer practical: 2hrs (of fun)
  - Installing R & RStudio
  - Creating an R project
  - Executing basic commands at the console
  - Executing commands from an R script
  - Reading and manipulating data
  - Plotting features

What is R?

What is R?

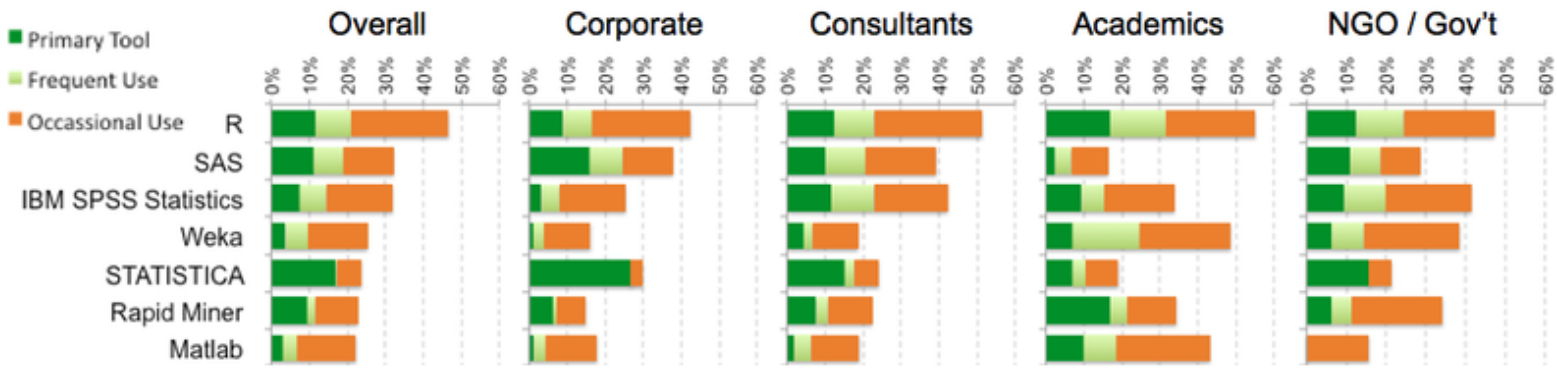
- R is a programming **language** and **environment** for statistical computing and graphics
- Language
  - Highly customisable, flexible and powerful
  - R was originally developed at the University of Auckland, NZ
  - Open-source version of the S-PLUS statistical programming language
- Environment
  - Supports the integration of many processes into one single work-flow/pipeline
  - Extensible, with thousands of (free) packages available online

What is R?

- It is used widely in academic and commercial sectors

mozilla Thomas Cook okcupid The New York Times facebook Google NewScientist

- It is one of the most popular analytics softwares used among professional data analysts



What is R

- pros
  - Fully functional statistical programming environment
  - Open source
  - Cross platform: can be used on all popular operating systems
  - Excellent graphics capabilities
  - Thousands of (free) extension packages.*
  - Large online community of users and developers
  - Analysis frameworks can be saved in reproducible formats.
  - High level programming language
- cons
  - Steep learning curve (Lots of help available)
  - Minimal GUI capabilities
  - Analysing large data sets can be troublesome (`??bigmemory`)
  - Scripts cannot be compiled into stand alone .exe programs
  - Interpreted language (slow compared to compiled C++ etc.)
  - Thousands of (free) extension packages.*

What can you do in R?

What can you do in R?

- Simple arithmetic

```
5 * 5
[1] 25
```

- Advanced calculator

Formula

$I_n(Q; J = j) = -p_j log_e p_j + \sum_{i=1}^{K} \frac{p_{ij}}{K} log_e p_{ij}$

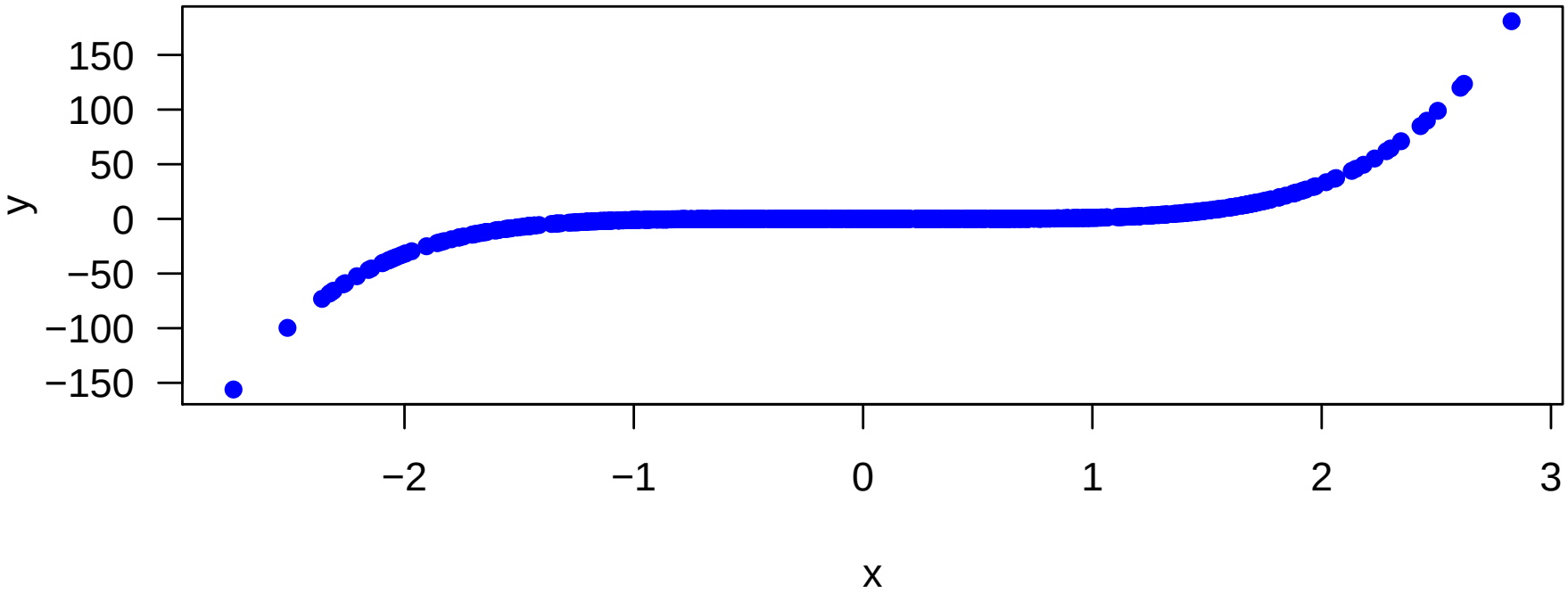
Code

```
In[j] <- (-(p[j])*log(p[j])) + sum((p[i, j]/k)*log(p[i, j]))
```

What can you do in R

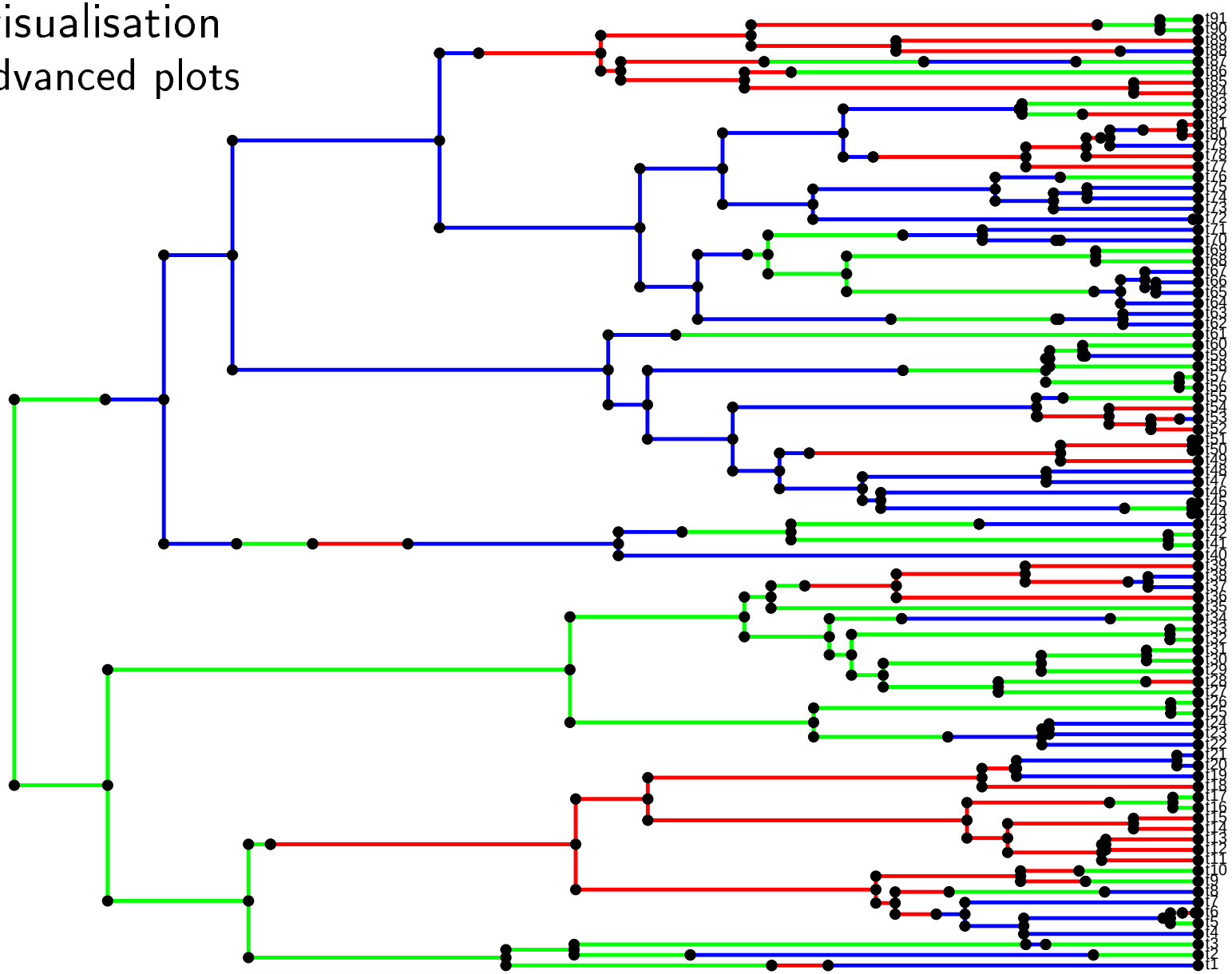
- Data analysis
  - General statistical tests
    (`t.test`, `anova`, `glm`, `chisq.test`, `lm`)
  - Specialised statistical packages
    (`abc`, `BSgenome`, `cluster`, `igraph`, `shapefiles`)

- Data Visualisation
  - Basic plots



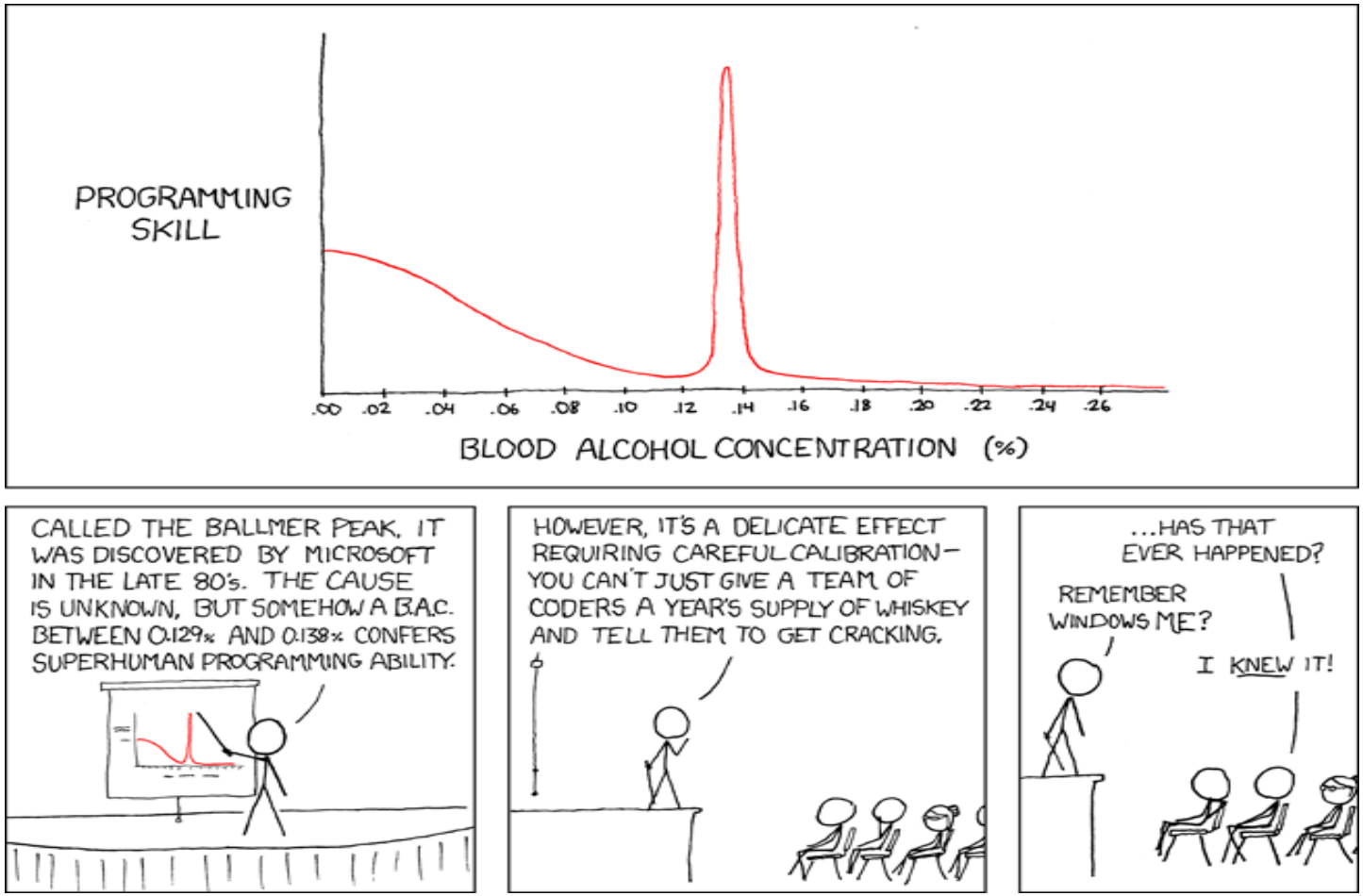
What R can do for you

- Data visualisation
  - Advanced plots



What R can do for you

- Having fun
  - Read XKCD comic strips



What can you do in R

- Designing web applications

Stocks app

Global Biodiversity Facility

Price trajectories

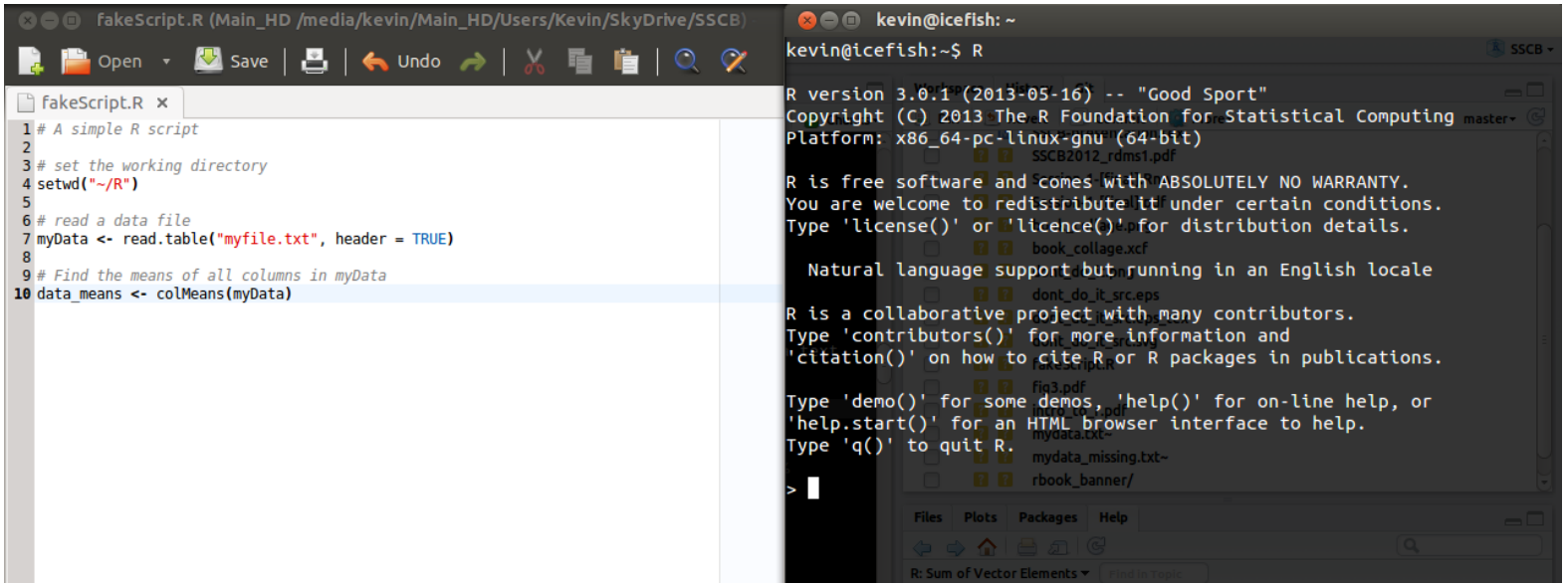
Gene Networks

Finance Showreel

R as a programming language

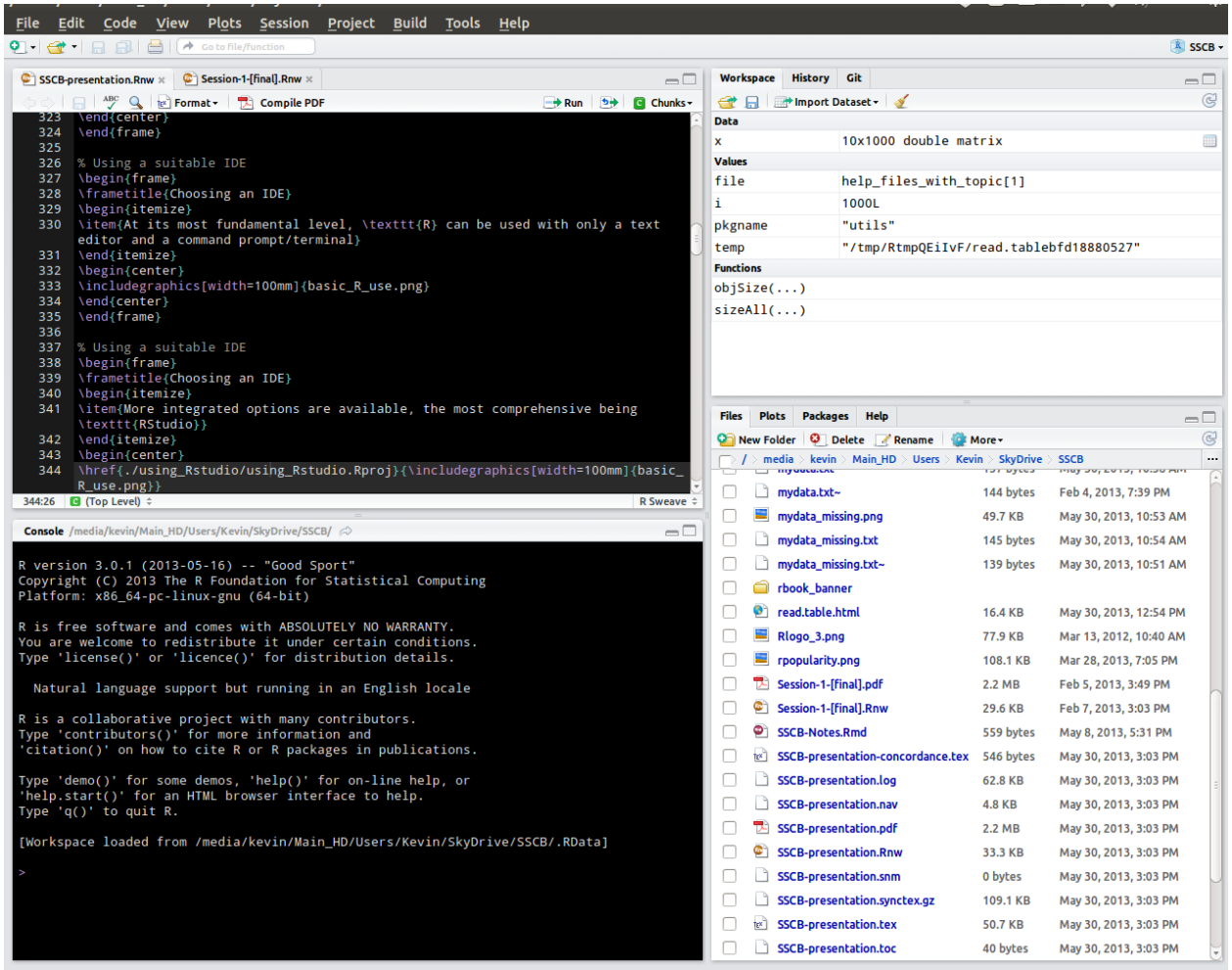
Choosing an IDE

- At its most fundamental level, R can be used with only a text editor and a command prompt/terminal



Choosing an IDE

- More integrated options are available, the most comprehensive being `RStudio`



The working directory

- The working directory is the directory level at which R will evaluate any calls to read or write data
- If a file of interest in present in the working directory, this file can be accessed by simply passing its name to a given function
- If the working directory is not set, either an absolute path, or a path relative to the current working directory must be given

```
# set the working directory
setwd("~/R_stuff")
# get the current working directory
getwd()
 [1] "~/R_stuff"
```

Finding help

- R has a built in help system

```
# Standard syntax
help(FUN)  # FUN = some function (e.g. 'sum')
# fuzzy search
help.search(pattern)
# or short-hand syntax
?FUN
??FUN
# example
?read.table
```

read.table help file

- The web is your friend

| UCLA R resources page | Search engine tailored to R | stackoverflow | Twitter #rstats | Comprehensive R Archive Network | R bloggers aggrigator site | The R podcast | The R Journal (peer reviewed) | Quick R | R programming wikibook |

Finding help

- Many excellent books

Basic operations

- Using R as a simple calculator

```
# The R console is interactive
5 + 5

[1] 10
```

- Using R as a (less) simple calculator

```
(1 + 1/10%%3)^100

[1] 1.268e+30
```

- Testing logicals

```
(1 + 1/10%%3)^100 == 2^100

[1] TRUE
```

Basic operations

- Assigning values to variables

```
# Using assignment operator '<-' ('=' in other languages)
x <- (1 + 1/10%%3)^100
y <- 2^100
x == y

[1] TRUE
```

- Common mathematical operators

```
^ or **              # power: 2^10 = 1024 = 2**10
* and /              # Multiplication and division
+ and -              # Addition and subtraction
%/%                  # Integer division: 10%/%3 = 3
%*%                  # Conformable matrix multiplication: x %*% y
%%                   # Modulo: 10%%3 = 1
()                   # Parentheses: specify operation order
```

Basic operations

- Common built in mathematical functions

```
sin(), cos(), tan(), exp(), log(), log10(),
sqrt(), sum(), prod(), floor(), ceiling(),
round(), abs(), acos(), atan(), factorial(), ...
```

- Examples

```
# find the product of a range of numbers
prod(1,2,3,4,5)

[1] 120

prod(1:5) == factorial(5)

[1] TRUE
```

Data types in R

- Values in R can be said to have a **class** or **mode**

```
typeof(5.2)                    # numeric class: double

[1] "double"

typeof("ABC")                  # character class: is.character()

[1] "character"

typeof(as.integer(5.2))        # numeric class: integer

[1] "integer"

typeof(TRUE)                   # Boolean: is.logical()

[1] "logical"

is.na(NA)                      # Missing data

[1] TRUE
```

Data structures

- Virtually everything in R is an **object/variable**
  - Facilitates Object-Oriented Programming

- Simple data structure (all vectors with different dimensionality!)

`matrix()`	All elements must be the same **mode**
`vector()`	All elements must be the same **mode**
`array()`	All elements must be the same **mode**

- Sophisticated data structures (Data analysis and manipulation!)

`factor()`	Special vector for categorical data **mode**
`list()`	Can have elements of different **mode**
`data.frame()`	Can have elements of different **mode**

Data structures: Examples

- Create a vector

```
x <- c(11, 12, 13, 14, 15)  # create a numeric vector
x                           # print x

[1] 11 12 13 14 15

x <- 11:15                  # using colon expansion
```

- What about strings?

```
y <- c("Kevin", "Keenan")   # create a character vector
y                           # print y

[1] "Kevin"  "Keenan"
```

- What about mixtures? (Beware coercion!!!)

```
z <- c("Kevin", "Keenan", 26)   # create mixed variable
z

[1] "Kevin"  "Keenan" "26"
```

Data structures: Examples

- For variables of different **mode**, we can use a `list`

```
z <- list(first = "Kevin", second = "Keenan",
          age = 26)
z
```

```
$first
[1] "Kevin"

$second
[1] "Keenan"

$age
[1] 26
```

- Make sure the elements have maintained their modes

```
is.character(z$first)
```

```
[1] TRUE
```

```
is.numeric(z$age)
```

```
[1] TRUE
```

Data structures: Examples

- A `matrix` is a `vector` with two **dimensions**, while an `array` is a matrix with more than two **dimensions**.

```
x_vect <- seq(from = 0.1, to = 10, by = 0.1)
length(x_vect)
```

```
[1] 100
```

```
x_mat <- matrix(x_vect, ncol = 10)
x_mat
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]  0.1  1.1  2.1  3.1  4.1  5.1  6.1  7.1  8.1   9.1
 [2,]  0.2  1.2  2.2  3.2  4.2  5.2  6.2  7.2  8.2   9.2
 [3,]  0.3  1.3  2.3  3.3  4.3  5.3  6.3  7.3  8.3   9.3
 [4,]  0.4  1.4  2.4  3.4  4.4  5.4  6.4  7.4  8.4   9.4
 [5,]  0.5  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5   9.5
 [6,]  0.6  1.6  2.6  3.6  4.6  5.6  6.6  7.6  8.6   9.6
 [7,]  0.7  1.7  2.7  3.7  4.7  5.7  6.7  7.7  8.7   9.7
 [8,]  0.8  1.8  2.8  3.8  4.8  5.8  6.8  7.8  8.8   9.8
 [9,]  0.9  1.9  2.9  3.9  4.9  5.9  6.9  7.9  8.9   9.9
[10,]  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0
```

Indexing/sub-setting

- Unlike languages such as `python` or `perl`, `R` indexing begins at 1

```
x[1]                    # print the first element of x

[1] 11

x[c(1, 4, 5)]           # print a range of elements

[1] 11 14 15

x[-3]                   # print all but the 3rd element

[1] 11 12 14 15

x[1] + x[2]             # Add the first and second elements

[1] 23

y <- x[-4]              # create a new vector
y

[1] 11 12 13 15
```

Indexing/sub-setting

- Because a `matrix` is a vector with two dimensions, each element has a pair of index 'coordinates'
- The general index format is as below:

```
matrix[row, column]
```

- If we wanted to extract the 5^{th} element of the 3^{rd} column, we would write:

```
x_mat[5, 3]

[1] 2.5
```

Indexing/sub-setting

- `list` indexing can be done through brackets (`[[]]`) or the '$' extract operator

```
# Create a list for the number of offspring per pet
offspring <- list(dog = c(2, 5, 3, 8), cat = c(12, 6, 7, 4),
                  snake = c(43, 35, 23, 10))
offspring


$dog
[1] 2 5 3 8

$cat
[1] 12  6  7  4

$snake
[1] 43 35 23 10

offspring$cat == offspring[[2]]   # Compare indexing methods


[1] TRUE TRUE TRUE TRUE
```

Why lists?

- The generality of the definition of a `list` makes it an extremely flexible data structure for statistical analyses
  - A generic container for other objects (e.g. matrices, vectors, data.frames)

```
# Create a complex list
myList <- list(x = matrix(c(offspring$dog, offspring$cat,
                            offspring$snake), ncol = 3),
               y = 10,
               z = offspring)
myList$x                    # print element 'x' of the list
```

```
     [,1] [,2] [,3]
[1,]    2   12   43
[2,]    5    6   35
[3,]    3    7   23
[4,]    8    4   10
```

```
myList$z$dog   == myList$x[, 1]
```

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

Why lists?

- Extracting multiple items from a list can be done in two ways

```
# Create a complex list
myList[c("x", "y")]                    # by name


$x
     [,1] [,2] [,3]
[1,]    2   12   43
[2,]    5    6   35
[3,]    3    7   23
[4,]    8    4   10

$y
[1] 10

myList[1:2]                            # by index


$x
     [,1] [,2] [,3]
[1,]    2   12   43
[2,]    5    6   35
[3,]    3    7   23
[4,]    8    4   10

$y
[1] 10
```

The data.frame: a special list

- The `data.frame` is the most used data structure in `R` for statistical analyses
- It is essentially a list with two dimensions

```
# Create a complex list
myDF <- data.frame(pet = c(rep("dog", 4), rep("cat", 4),
                           rep("snake", 4)),
                   offspring = unlist(offspring))
myDF
```

```
        pet offspring
dog1    dog         2
dog2    dog         5
dog3    dog         3
dog4    dog         8
cat1    cat        12
cat2    cat         6
cat3    cat         7
cat4    cat         4
snake1 snake       43
snake2 snake       35
snake3 snake       23
snake4 snake       10
```

The data.frame: a special list

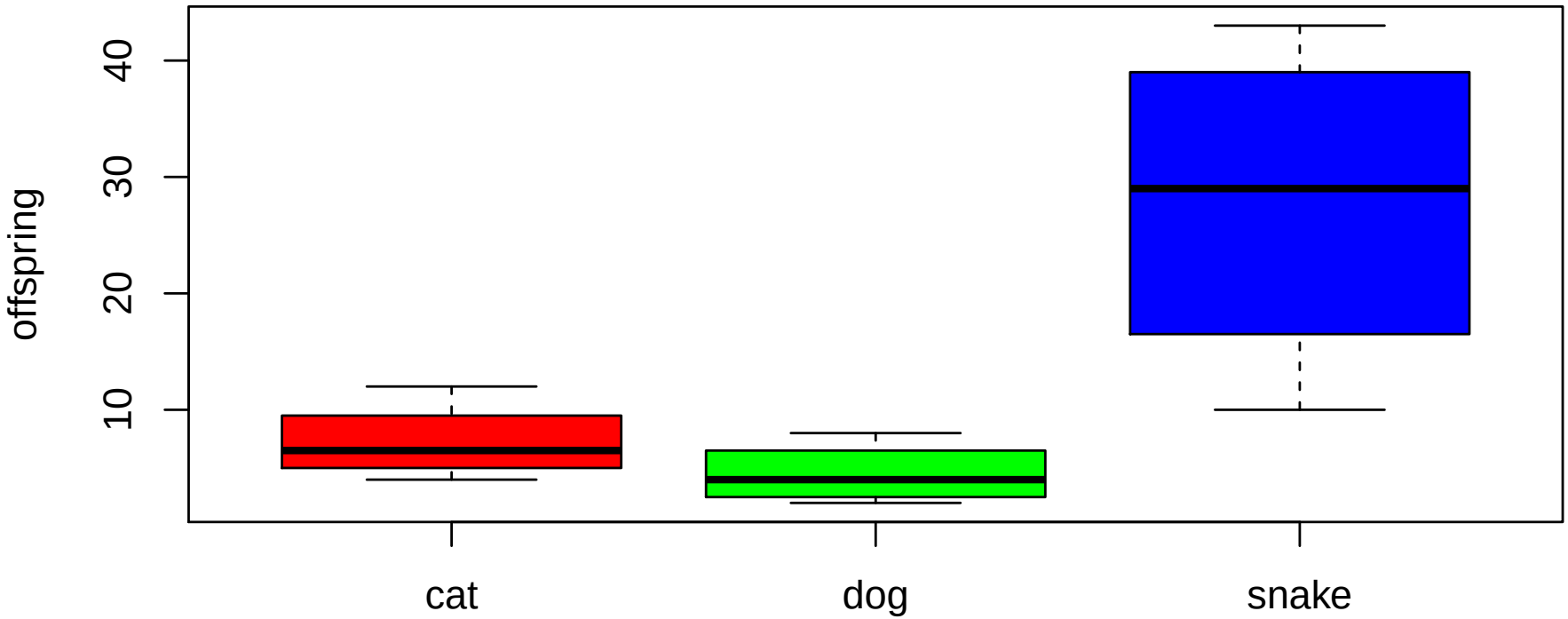
- The pet variable in myDF is a factor
  - A special vector for categorical data

```
is.factor(myDF$pet)          # check if pet is a factor

[1] TRUE

levels(myDF$pet)             # check the levels (categories) of pet

[1] "cat"   "dog"   "snake"

tapply(myDF$offspring, myDF$pet, FUN = mean)

  cat   dog snake
 7.25  4.50 27.75
```

The data.frame: a special list

- Data frames containing factors make plotting categorical data simple

```
plot(myDF, col = rainbow(3))    # boxplot of offspring per pet
```



Real data in R

- To read data from a file into R, there are a range of simple functions

```
read.csv()          # read a comma seperated file
read.table()        # read a file in table format
read.delim()        # read a file with TAB delimited fields
read.fwf()          # read a fixed width format table
read.DIF()          # read data interchange format file
scan()              # C level: foundation for most other read functions
readLines()         # Read individual lines from a file
```

- The most commonly used function is `read.table`

	A	B	C	D
1	Name	Height	Weight	BMI
2	Steve	1.78	75.2	23.73
3	Emily	1.56	120.4	49.47
4	George	2.11	91.3	20.51
5	Nicola	1.65	45.6	16.75
6	Pierre	1.95	86.3	22.70

Real data in R

- To read a spreadsheet table, it is generally sensible to convert it to a text file or .csv file
  - This avoids any problems with multi-sheet workbooks and odd text encoding

```
my_data <- read.table("mydata.txt", header = TRUE)
print(my_data)                          # look at the table in R

    Name Height Weight   BMI
1  Steve   1.78   75.2 23.73
2  Emily   1.56  120.4 49.47
3 George   2.11   91.3 20.51
4 Nicola   1.65   45.6 16.75
5 Pierre   1.95   86.3 22.70

is.data.frame(my_data)

[1] TRUE
```

Real data in R

- Some simple manipulations of my_data

```
str(my_data)                # check the structure of my_data

'data.frame': 5 obs. of  4 variables:
 $ Name  : Factor w/ 5 levels "Emily","George",..: 5 1 2 3 4
 $ Height: num  1.78 1.56 2.11 1.65 1.95
 $ Weight: num  75.2 120.4 91.3 45.6 86.3
 $ BMI   : num  23.7 49.5 20.5 16.8 22.7

names(my_data)              # get variable names

[1] "Name"   "Height" "Weight" "BMI"

names(my_data) <- c("nms", "hgt", "wgt", "bmi")  # change names
names(my_data)

[1] "nms" "hgt" "wgt" "bmi"

mean(my_data$hgt)           # get the mean of heights

[1] 1.81
```

Missing data

- Imagine that we didn't have any height information for Emily

	A	B	C	D
1	Name	Height	Weight	BMI
2	Steve	1.78	75.2	23.73
3	Emily	-9999	120.4	-9999
4	George	2.11	91.3	20.51
5	Nicola	1.65	45.6	16.75
6	Pierre	1.95	86.3	22.70

```
msng_data <- read.table("mydata_missing.txt", header = TRUE)
print(msng_data)

    Name   Height Weight       BMI
1  Steve     1.78   75.2     23.73
2  Emily -9999.00  120.4 -9999.00
3 George     2.11   91.3     20.51
4 Nicola     1.65   45.6     16.75
5 Pierre     1.95   86.3     22.70

mean(msng_data$Height)     # R is not treating -9999 as missing

[1] -1998
```

Missing data

- The special value NA is reserved for missing data in R
  - We need to replace our missing values (-9999) with NA's

```
msng_data[msng_data == -9999] <- NA      # replace missing data
print(msng_data)

    Name Height Weight   BMI
1  Steve   1.78   75.2 23.73
2  Emily     NA  120.4    NA
3 George   2.11   91.3 20.51
4 Nicola   1.65   45.6 16.75
5 Pierre   1.95   86.3 22.70

mean(msng_data$Height, na.rm = TRUE)     # calculate the mean

[1] 1.873
```

User defined functions

- User defined functions in R are similar to **modules**, **subroutines** or **procedures** in other languages
- They take the general form below:

```
funName <- function(arg1, arg2, ...){
  expression1
  expression2
  ...
  return(output)
}
```

- A simple example: calculate the square of a number

```
# Define the function
sqrFun <- function(x){
  x_sqr <- x^2
  return(x_sqr)
}
sqrFun(10)          # test the function

[1] 100
```

User defined functions

- A more advanced `function` using **conditional control flow**
  - Allow for user defined operation on a vector (e.g. sum or product)

```
# define the new complex function
newFun <- function(data, operation = "sum"){
  if(operation == "sum"){
    output <- sum(data)
  } else if(operation == "product"){
    output <- prod(data)
  }
  return(output)
}
newFun(data = offspring$dog, operation = "sum")

[1] 18

newFun(data = offspring$dog, operation = "product")

[1] 240
```

User defined functions

- Test the function

```
sum_dog <- newFun(data = offspring$dog, operation = "sum")
prod_dog <- newFun(data = offspring$dog, operation = "product")
sum_dog                                  # print sum results

[1] 18

prod_dog                                 # print product results

[1] 240

sum_dog == sum(offspring$dog)            # compare function sums

[1] TRUE

prod_dog == prod(offspring$dog)          # compare function products

[1] TRUE
```

Iterative processes in R

- One of the most useful features of a programming language is the ability to do many things, multiple times, quickly.
- R allows us to achieve this with the help of 'loops'
- There are **three** main loop structures in R.
  - `for loop`
  - `while loop`
  - `repeat loop`

```
?`for`
?`while`
?`repeat`
```

The `for` loop

Say we have a variable, `x`, as follows:

```
x <- seq(from = 10, to = 100, by = 10)
```

To iteratively print each element of `x` to the console, we do the following:

```
for(i in x){
  print(i)
}
```

```
[1] 10
[1] 20
[1] 30
[1] 40
[1] 50
[1] 60
[1] 70
[1] 80
[1] 90
[1] 100
```

The `while` loop

To do the same thing using the `while` loop:

```
# Create an initial value for x
x <- 10
# Construct the while loop
while(x <= 100){
  print(x)
  x <- x + 10
}
```

```
[1] 10
[1] 20
[1] 30
[1] 40
[1] 50
[1] 60
[1] 70
[1] 80
[1] 90
[1] 100
```

The `repeat` loop

To do the same thing using the `repeat` loop:

```
# create a starting variable
x <- 10
# repeat loop construct
repeat {
    if (x > 100) {
        break
    } else {
        print(x)
        x <- x + 10
    }
}
```

```
[1] 10
[1] 20
[1] 30
[1] 40
[1] 50
[1] 60
[1] 70
[1] 80
[1] 90
[1] 100
```

R as statistical package

Base R functions

- Base R is the standard distribution of R without additional **packages** installed
- It contains many useful functions for general statistical analysis

```
t.test()                # student's t-test
chisq.test()            # chi-square tests
glm()                   # general/generalised linear models
cor.test()              # correlations
lm()                    # Logistic regression
aov()                   # ANOVA
princomp()              # PCA
kmeans()                # Multivariate clustering
hclust()                # Hierarchical clustering
```

- In many cases there are multiple specialised function to do similar analyses

Function help

- All functions should have help files associated with them to aid usage

```
?t.test
```

t.test help file

- These help files can take some getting used to
- Sometimes a web search will provide nice easy to understand worked examples

http://www.statmethods.net/stats/ttest.html

Extending R with packages

- In many fields of research, general statistical methods just don't cut it
- R has a vibrant community of developers leading to the availability of ≈5000 add-on packages
  - CRAN
  - Bioconductor

Further reading

Crawley, M.J. (2013). The R Book. Wiley

Jones, O., Maillardet, R., & Robinson, A. (2009). Introduction to scientific programming and simulation using R. CRC Press

Matloff, N. (2011). The art of R programming: a tour of statistical software design. No Starch Press.

Spector, P. (2008). Data manipulation with R. Springer: useR series

Reproducibility

```
R version 3.0.1 (2013-05-16)
Platform: x86_64-w64-mingw32/x64 (64-bit)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] knitr_1.4.1

loaded via a namespace (and not attached):
[1] digest_0.6.3   evaluate_0.4.7 formatR_0.9    highr_0.2.1
[5] stringr_0.6.2  tools_3.0.1
```

Practical (FUN) time