Introduction to Solving Biological Problems Using R - Day 1

Mark Dunning. Original material by Robert Stojnić, Laurent Gatto, Rob Foy John Davey, Dávid Molnár and Ian Roberts

08/09/2014

Introduction to R and its environment

Data structures

R for data analysis

Programming techniques

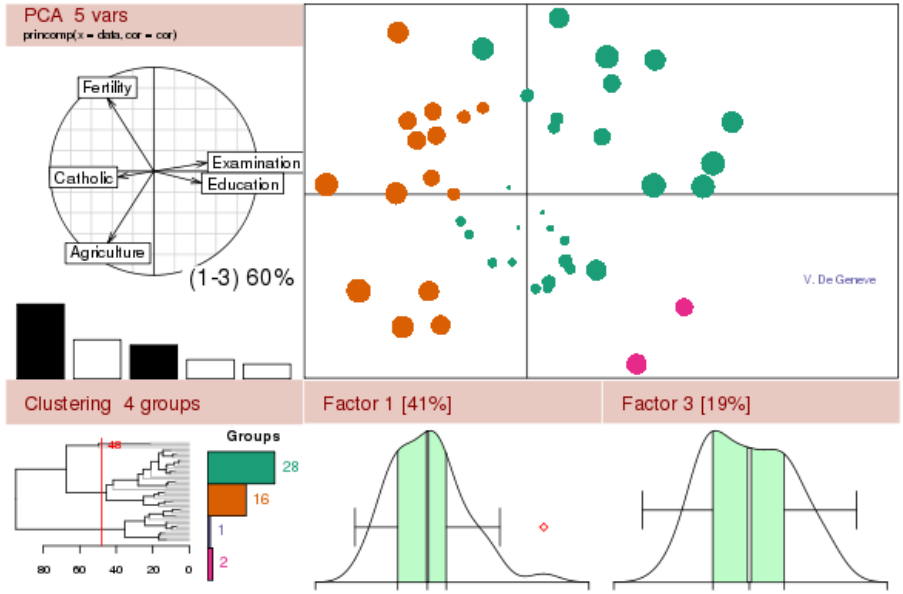
End of Day 1

Introduction to R and its environment

What's R?

- A statistical programming environment
  - based on 'S'
  - suited to high-level data analysis
- Open source and cross platform
- Extensive graphics capabilities
- Diverse range of add-on packages
- Active community of developers
- Thorough documentation

The R Project for Statistical Computing



About R
What is R?
Contributors
Screenshots
What's new?

Download, Packages
CRAN

R Project
Foundation
Members & Donors
Mailing Lists
Bug Tracking
Developer Page
Conferences
Search

Documentation
Manuals
FAQs
The R Journal
Wiki
Books
Certification
Other

Misc
Bioconductor
Related Projects
User Groups
Links

Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To **download R**, please choose your preferred CRAN mirror.
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

News:

- **R version 3.1.1** (Sock it to Me) has been released on 2014-07-10.
- **R version 3.0.3** (Warm Puppy) has been released on 2014-03-06.
- **The R Journal Vol.5/2** is available.
- **useR! 2013**, took place at the University of Castilla-La Mancha, Albacete, Spain, July 10-12 2013.

R in the New York Times

The New York Times

Business Computing

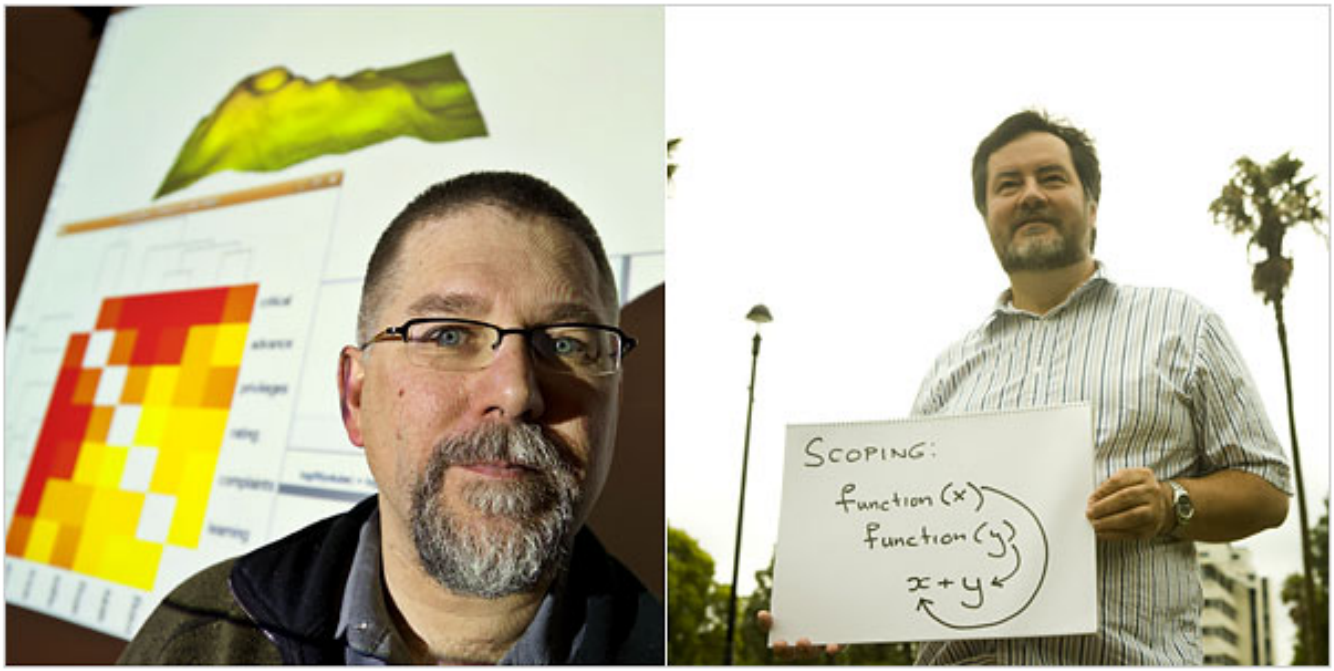
WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION

Search Technology | Go

Inside Technology: Internet | Start-Ups | Business Computing | Companies

Bits Blo

Data Analysts Captivated by R's Power



Stuart Isett for The New York Times

R first appeared in 1996, when the statistics professors Robert Gentleman, left, and Ross Ihaka released the code as a free software package.

R in Nature

nature International weekly journal of science

NATURE | TOOLBOX

Programming tools: Adventures with R

A guide to the popular, free statistics and visualization software that gives scientists control of their own data analysis.

Sylvia Tippmann

29 December 2014

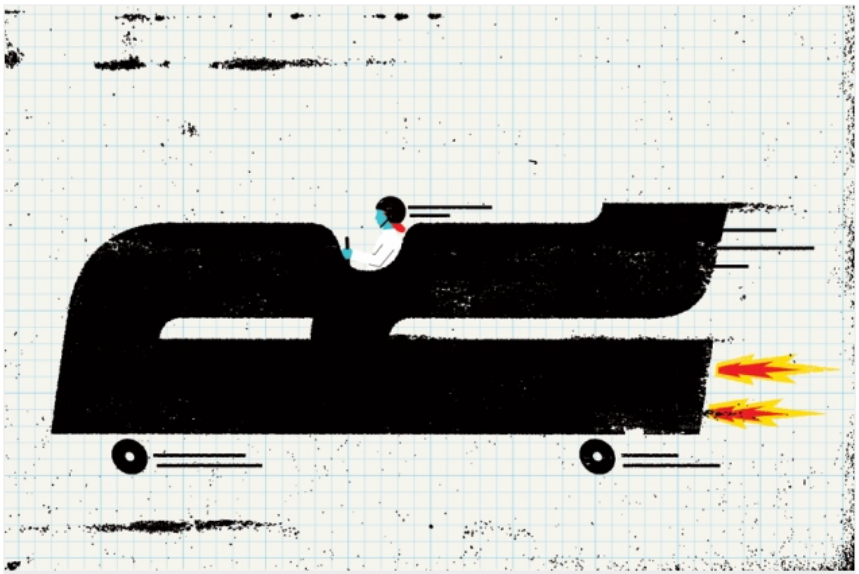


Illustration by The Project Twins

Various platforms supported

- Release 3.2.0 (April 2015)
  - Base package and Contributed packages (general purpose extras)
  - 6619 available packages
- Download from `http://mirrors.ebi.ac.uk/CRAN/`
- Windows, Mac and Linux versions available
- Executed using command line, or a graphical user interface (GUI)
- On this course, we use the RStudio GUI (www.rstudio.com)
- Everything you need is installed on the training machines
- If you are using your own machine, download both R and RStudio

Getting started

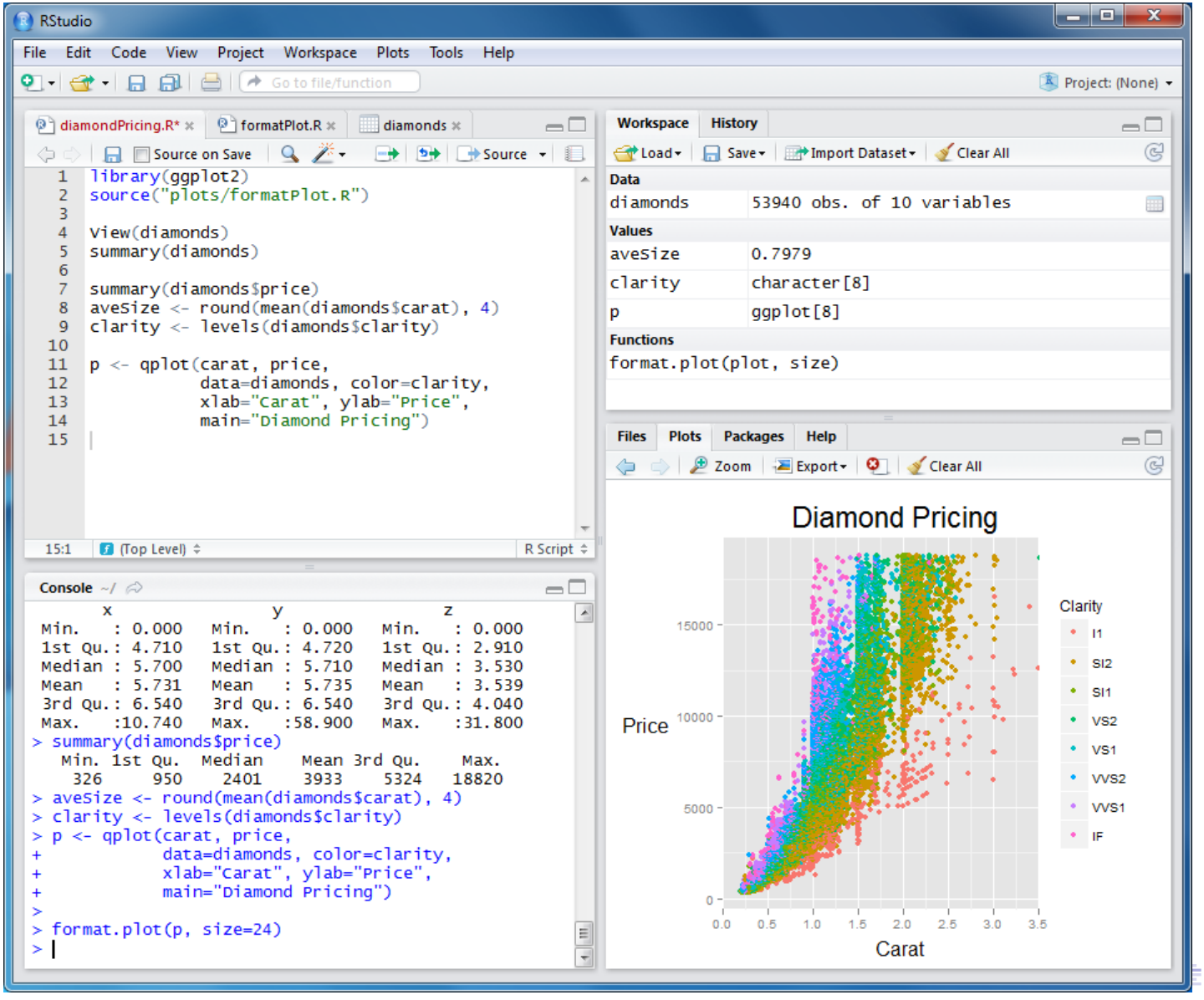
R is a program which, once installed on your system, can be launched and is immediately ready to take input directly from the user

There are two ways to launch R:

- From the command line (particularly useful if you're quite familiar with Linux)
- As an application called RStudio (very good for beginners)

Launching R Using RStudio

To launch RStudio, find the RStudio icon in the menu bar on the left of the screen and double-click



The Working Directory (wd)

- Like many programs R has a concept of a working directory (wd)
- It is the place where R will look for files to execute and where it will save files, by default
- For this course we need to set the working directory to the location of the course scripts
- At the command prompt in the terminal or in RStudio console type:

```
setwd("R_course/Day_1_scripts")
```

- Alternatively in RStudio use the mouse and browse to the directory location
- Session → Set Working Directory → Choose Directory...

Basic concepts in R - command line calculation

- The command line can be used as a calculator. Type:

```
2 + 2
```

```
## [1] 4
```

```
20/5 - sqrt(25) + 3^2
```

```
## [1] 8
```

```
sin(pi/2)
```

```
## [1] 1
```

Note: The number in the square brackets is an indicator of the position in the output. In this case the output is a 'vector' of length 1 (i.e. a single number). More on vectors coming up. . .

Basic concepts in R - variables

- A variable is a letter or word which takes (or contains) a value. We use the assignment 'operator', <-

```
x <- 10

x
```

```
## [1] 10
```

```
myNumber <- 25
myNumber
```

```
## [1] 25
```

Basic concepts in R - variables

- We can perform arithmetic on variables:

```
sqrt(myNumber)
```

```
## [1] 5
```

We can add variables together:

```
x + myNumber
```

```
## [1] 35
```

Basic concepts in R - variables

- We can change the value of an existing variable:

```
x <- 21
x
```

```
## [1] 21
```

- We can set one variable to equal the value of another variable

```
x <- myNumber
x
```

```
## [1] 25
```

Basic concepts in R - variables

- We can modify the contents of a variable

```
myNumber <- myNumber + sqrt(16)
myNumber
```

```
## [1] 29
```

Basic concepts in R - functions

- **Functions** in R perform operations on **arguments** (the inputs(s) to the function). We have already used

```
sin(x)
```

this returns the sine of x. In this case the function has one argument, **x**. Arguments are always contained in parentheses., i.e. curved brackets **()**, separated by commas

Basic concepts in R - functions

- Try these:

```
sum(3,4,5,6)
```

```
## [1] 18
```

```
max(3,4,5,6)
```

```
## [1] 6
```

```
min(3,4,5,6)
```

```
## [1] 3
```

Basic concepts in R - functions

- Arguments can be named or unnamed, but if they are unnamed they must be ordered (we will see later how to find the right order)

```
seq(from =2, to=10, by=2)
```

```
## [1]  2  4  6  8 10
```

```
seq(2,10,2)
```

```
## [1]  2  4  6  8 10
```

Basic concepts in R - vectors

- The basic data structure in R is a **vector** - an ordered collection of values. R even treats single values as 1-element vectors. The function **c()** *combines* its arguments into a vector:

```
x <- c(3,4,5,6)
x
```

```
## [1] 3 4 5 6
```

- The square brackets **[]** indicate the position within the vector (the **index**). We can extract individual elements by using the **[]** notation

```
x[1]
```

```
## [1] 3
```

```
x[4]
```

Basic concepts in R - vectors

- We can even put a vector inside the square brackets (*vector indexing*)

```
y <- c(2,3)
x[y]
```

```
## [1] 4 5
```

Basic concepts in R - vectors

- There are a number of shortcuts to create a vector. Instead of:

```
x <- c(3,4,5,6,7,8,9,10,11,12)
```

- we can write

```
x <- 3:12
x
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12
```

Basic concepts in R - vectors

- or we can use the **seq()** function, which returns a vector

```
x<- seq(2,10,2)
x
```

```
## [1]  2  4  6  8 10
```

```
x <- seq(2,10,length.out=7)
x
```

```
## [1]  2.000000  3.333333  4.666667  6.000000  7.333333
## [6]  8.666667 10.000000
```

Basic concepts in R - vectors

- or we can use the **rep()** function

```
y <- rep(3,5)
y
```

```
## [1] 3 3 3 3 3
```

```
y <- rep(1:3,5)
y
```

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

Basic concepts in R - vectors

- We have seen some ways of extracting elements of a vector. We can use these shortcuts to make things easier (or more complex!)

```
x <- 3:12
x[3:7]
```

```
## [1] 5 6 7 8 9
```

```
x[seq(2,6,2)]
```

```
## [1] 4 6 8
```

```
x[rep(3,2)]
```

```
## [1] 5 5
```

Basic concepts in R - vectors

- We can add an element to a vector

```
y <- c(x, 1)
y
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12  1
```

We can glue vectors together

```
z <- c(x,y)
z
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12  3  4  5  6  7  8  9
## [19] 11 12  1
```

Basic concepts in R - vectors

- We can remove element(s) from a vector

```
x <- 3:12
x[-3]
```

```
## [1]  3  4  6  7  8  9 10 11 12
```

```
x[-(5:7)]
```

```
## [1]  3  4  5  6 10 11 12
```

```
x[-seq(2,6,2)]
```

```
## [1]  3  5  7  9 10 11 12
```

Basic concepts in R - vectors

- Finally, we can modify the contents of a vector

```
x[6] <- 4
x
```

```
##  [1]  3  4  5  6  7  4  9 10 11 12
```

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

```
##  [1]  3  4  1  1  1  4  9 10 11 12
```

Remember! **Square** brackets for indexing [], **parentheses** for function arguments ().

Basic concepts in R - vector arithmetic

- When applying all standard arithmetic operations to vectors, application is element-wise

```
x <- 1:10
y <- x*2
y
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
z <- x^2
z
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Basic concepts in R - vector arithmetic

- Adding two vectors

```
y + z
```

```
##  [1]   3   8  15  24  35  48  63  80  99 120
```

If vectors are not the same length, the shorter one will be recycled:

```
x + 1:2
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

Basic concepts in R - vector arithmetic

But be careful if the vector lengths aren't factors of each other:

```
x + 1:3
```

```
## Warning in x + 1:3: longer object length is not a
## multiple of shorter object length
```

```
##  [1]  2  4  6  5  7  9  8 10 12 11
```

Basic concepts in R - Character vectors and naming

- All the vectors we have seen so far have contained numbers, but we can also store strings in vectors – this is called a **character** vector.

```
gene.names <- c("Pax6","Beta-actin","FoxP2","Hox9")
```

We can name elements of vectors using the `names` function, which can be useful to keep track of the meaning of our data:

```
gene.expression <- c(0,3.2,1.2,-2)
gene.expression
```

```
## [1]  0.0  3.2  1.2 -2.0
```

Basic concepts in R - Character vectors and naming

```
names(gene.expression)<-gene.names
gene.expression
```

```
##        Pax6 Beta-actin       FoxP2        Hox9
##         0.0         3.2         1.2        -2.0
```

- We can also use the `names` function to get a vector of the names of an object:

```
names(gene.expression)
```

```
## [1] "Pax6"        "Beta-actin" "FoxP2"
## [4] "Hox9"
```

Exercise: genes and genomes

- Let's try some vector arithmetic. Here are the genome lengths and number of protein coding genes for several model organisms:

Geno	me size (Mb) Prot	ein coding genes
Homo sapiens	3,102	20,774
Mus musculus	2,731	23,139
Drosophila melanogaster	169	13,937
Caenorhabditis elegans	100	20,532
Saccharomyces cerevisiae	12	6,692

Exercise: genes and genomes

- Create *genome.size* and *coding.genes* vectors to hold the data in each column using the c function. Create a *species.name* vector and use this vector to name the values in the other two vectors
- Let's assume a coding gene has an average length of 1.5 kilobases. On average, how many base pairs of each genome is made of coding genes? Create a new vector to record this called *coding.bases*
- What percentage of each genome is made up of protein coding genes? Use your *coding.bases* and *genome.size* vectors to calculate this. (See earlier slides for how to do division in R.)
- How many times more bases are used for coding in the human genome compared to the yeast genome? How many times more bases are in the human genome in total compared to the yeast genome? Look up indices of your vectors to find out.

Answers to genome exercise

```
genome.size<-c(3102,2731,169,100,12)
coding.genes<-c(20774,23139,13937,20532,6692)
species.name<-c("H. sapiens","M. musculus",
                "D. melanogaster","C. elegans",
                "S. cerevisiae")
names(genome.size)<-species.name
names(coding.genes)<-species.name
```

To calculate the number of coding bases, we need to use the same scale as we used for genome size: 1.5 kilobases is 0.0015 Megabases.

```
coding.bases<-coding.genes*0.0015
coding.bases
```

```
##       H. sapiens     M. musculus D. melanogaster
##          31.1610         34.7085         20.9055
##       C. elegans   S. cerevisiae
##          30.7980         10.0380
```

Answers to genome exercise

- To calculate the percentage of coding bases in each genome:

```
coding.pc<-coding.bases/genome.size*100
coding.pc
```

```
##       H. sapiens     M. musculus D. melanogaster
##         1.004545        1.270908       12.370118
##       C. elegans   S. cerevisiae
##        30.798000       83.650000
```

Answers to genome exercise

To compare human to yeast:

```
coding.bases[1]/coding.bases[5]
```

```
## H. sapiens
##   3.104304
```

```
genome.size[1]/genome.size[5]
```

```
## H. sapiens
##      258.5
```

Answers to genome exercise

- Note that if a new vector is created using a named vector, the names are usually carried across to the new vector. Sometimes this is what we want (as for coding.pc) but sometimes it is not (when we are comparing human to yeast). We can remove names by setting them to the special NULL value:

```
names(coding.pc)<-NULL
coding.pc
```

```
## [1]  1.004545  1.270908 12.370118 30.798000
## [5] 83.650000
```

Writing scripts with RStudio

- Typing lots of commands directly to R can be tedious. A better way is to write the commands to a file and then load it into R.
- Click on **File** -> **New** in Rstudio
- Type in some R code, e.g.

```
x <- 2 + 2
print(x)
```

- Click on **Run** to execute the current line, and **Source** to execute the whole script

Sourcing can also be performed manually with `source("myScript.R")`

Getting help

- To get help on any R function, type `?` followed by the function name. For example:

```
?seq
```

- This retrieves the syntax and arguments for the function. You can see the default order of arguments here. The help page also tells you which **package** it belongs to.
- There will typically be example usage, which you can test using the `example` function:

```
example(seq)
```

- If you can't remember the exact name type `??` followed by your guess. R will return a list of possibilities

```
??plot
```

Interacting with the R console

- R console symbols
  - ; end of line (Enables multiple commands to be placed on one line of text)
  - # comment (indicates text is a comment and not executed)
  - + command line wrap (R is waiting for you to complete an expression)
- *Ctrl-c* or *escape* to clear input line and try again
- *Ctrl-l* to clear window
- Use the *TAB* key for command auto completion
- Use up and down arrows to scroll through the command history

R packages

- R comes ready loaded with various libraries of functions called packages. e.g. the function `sum()` is in the base package and `sd()`, which calculates the standard deviation of a vector, is in the `stats` package
- There are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called *repositories*

R packages

- The two repositories you will come across the most are
- The Comprehensive R Archive Network (CRAN)
- Bioconductor
- CRAN packages can be installed using **install.packages**
- Set the Bioconductor package download tool by typing

```
source("http://bioconductor.org/biocLite.R")
```

- Bioconductor packages are then installed with the **biocLite()** function

```
biocLite("PackageName")
```

Exercise: Install packages ggplot2 and DESeq

- ggplot2 is a commonly used graphics package
- Use install.packages() function to install it

```
install.packages("ggplot2")
```

- or in RStudio goto Tools → Install Packages... and type the package name
- DESeq is a BioConductor package (www.bioconductor.org)

```
source("http://www.bioconductor.org/biocLite.R")
biocLite("DESeq")
```

R packages

- R needs to be told to use the new functions from the installed packages

```
library(ggplot2) # loads ggplot functions
library(DESeq) # loads DESeq functions
library() # Lists all the packages you've got installed
```

Data structures

R is designed to handle experimental data

- Although the basic unit of R is a vector, we usually handle data in **data frames**
- A data frame is a set of observations of a set of variables – in other words, the outcome of an experiment.
- For example, we might want to analyse information about a set of patients. To start with, let's say we have ten patients and for each one we know their name, sex, age, weight and whether they give consent for their data to be made public

The patients data frame

- We are going to create a data frame called 'patients', which will have ten rows (observations) and seven columns (variables). The columns must all be equal lengths.

```
##    First_Name Second_Name       Full_Name    Sex
## 1        Adam       Jones     Adam Jones   Male
## 2         Eve      Parker     Eve Parker Female
## 3        John       Evans     John Evans   Male
## 4        Mary       Davis     Mary Davis Female
## 5       Peter       Baker    Peter Baker   Male
## 6        Paul     Daniels   Paul Daniels   Male
## 7      Joanna     Edwards Joanna Edwards Female
## 8     Matthew       Smith  Matthew Smith   Male
## 9       David     Roberts  David Roberts   Male
## 10      Sally      Wilson   Sally Wilson Female
##    Age Weight Consent
## 1   50   70.8    TRUE
## 2   21   67.9    TRUE
```

Character, numeric and logical data types

- Each column is a vector, like previous vectors we have seen, for example:

```
age<-c(50, 21, 35, 45, 28, 31, 42, 33, 57, 62)
weight<-c(70.8, 67.9, 75.3, 61.9, 72.4, 69.9, 63.5,
71.5, 73.2, 64.8)
```

We can define the names using character vectors:

```
firstName<- c("Adam", "Eve", "John", "Mary", "Peter",
"Paul", "Joanna", "Matthew", "David", "Sally")
 secondName<-c("Jones", "Parker", "Evans", "Davis",
"Baker","Daniels", "Edwards", "Smith", "Roberts", "Wilson")
```

We also have a new type of vector, the *logical* vector, which only contains the values `TRUE` and `FALSE`:

```
 consent<-c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,
FALSE,TRUE,FALSE,TRUE)
```

Character, numeric and logical data types

- Vectors can only contain one type of data; we cannot mix numbers, characters and logical values in the same vector. If we try this, R will convert everything to characters:

```
c(20, "a string", TRUE)
```

```
## [1] "20"       "a string" "TRUE"
```

We can see the type of a particular vector using the `mode` function:

```
mode(firstName)
```

```
## [1] "character"
```

```
mode(age)
```

```
## [1] "numeric"
```

```
mode(weight)
```

Factors

- Character vectors are fine for some variables, like names
- But sometimes we have categorical data and we want R to recognize this
- A factor is R's data structure for categorical data

```
sex<-c("Male", "Female", "Male", "Female", "Male", "Male",
"Male", "Male", "Female")
```

```
sex
```

```
##  [1] "Male"   "Female" "Male"   "Female"
##  [5] "Male"   "Male"   "Female" "Male"
##  [9] "Male"   "Female"
```

Factors

```
factor(sex)
```

```
##   [1] Male   Female Male   Female Male
##   [6] Male   Female Male   Male   Female
## Levels: Female Male
```

- R has converted the strings of the sex character vector into two **levels**, which are the categories in the data
- Note the values of this factor are not character strings, but levels
- We can use this factor to compare data for males and females

Creating a data frame (first attempt)

- We can construct a data frame from other objects

```
patients<-data.frame(firstName, secondName,
    paste(firstName,secondName),sex, age,
    weight, consent)
patients[1:3,]
```

```
##   firstName secondName
## 1      Adam      Jones
## 2       Eve     Parker
## 3      John      Evans
##   paste.firstName..secondName.    sex
## 1                   Adam Jones   Male
## 2                   Eve Parker Female
## 3                   John Evans   Male
##   age weight consent
## 1  50   70.8    TRUE
## 2  21   67.9    TRUE
```

Creating a data frame (first attempt)

- The `paste` function joins character vectors together
- We can access particular variables using the *dollar* operator

```
patients$age
```

```
##  [1] 50 21 35 45 28 31 42 33 57 62
```

Naming data frame variables

- R has inferred the names of our data frame variables from the names of the vectors or the commands (eg the paste command)
- We can name the variables after we have created a data frame using the `names` function, and we can use the same function to see the names

```
names(patients)<-c("First_Name", "Second_Name",
"Full_Name", "Sex", "Age",
"Weight", "Consent")
```

```
names(patients)
```

```
## [1] "First_Name"  "Second_Name"
## [3] "Full_Name"   "Sex"
## [5] "Age"         "Weight"
## [7] "Consent"
```

Naming data frame variables

- Or we can name the variables when we define the data frame

```
patients<-data.frame(First_Name=firstName,
  Second_Name=secondName,
  Full_Name=paste(firstName,secondName),
  Sex=sex, Age=age, Weight=weight, Consent=consent)
```

```
names(patients)
```

```
## [1] "First_Name"  "Second_Name"
## [3] "Full_Name"   "Sex"
## [5] "Age"         "Weight"
## [7] "Consent"
```

Factors in data frames

- When creating a data frame, R assumes all character vectors should be categorical variables and converts them to factors. This is not always what we want:

```
patients$First_Name
```

```
##  [1] Adam    Eve     John    Mary
##  [5] Peter   Paul    Joanna  Matthew
##  [9] David   Sally
## 10 Levels: Adam David Eve ... Sally
```

Factors in data frames

- We can avoid this by asking R not to treat strings as factors, and then explicitly stating when we want a factor by using `factor`

```
patients<-data.frame(First_Name=firstName,
      Second_Name=secondName,
      Full_Name=paste(firstName,secondName),
      Sex=factor(sex), Age=age,
Weight=weight, Consent=consent, stringsAsFactors=FALSE)
```

Factors in data frames

```
patients$Sex
```

```
##  [1] Male   Female Male   Female Male
##  [6] Male   Female Male   Male   Female
## Levels: Female Male
```

```
patients$First_Name
```

```
##  [1] "Adam"    "Eve"     "John"
##  [4] "Mary"    "Peter"   "Paul"
##  [7] "Joanna"  "Matthew" "David"
## [10] "Sally"
```

Matrices

- Data frames are R's speciality, but R also handles matrices:

```
e <- matrix(1:10, nrow=5, ncol=2)
e
```

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

```
f <- matrix(1:10, nrow=2, ncol=5)
f
```

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

Matrices

```
f %*% e
```

```
##      [,1] [,2]
## [1,]   95  220
## [2,]  110  260
```

The %*% operator is the matrix multiplication operator, not the standard multiplication operator

Lists

- We have seen that vectors can only hold data of one type. How can we store data of multiple types? Or vectors of different lengths in one object?
- We can use lists. A list can contain objects of any type

```
one.to.ten <- 1:10
uniform.mat <- matrix(runif(100),ncol=10,nrow=10)
year.to.october <- data.frame(one.to.ten, month.name[1:10])
myList<-list( ls.obj.1=one.to.ten, ls.obj.2=uniform.mat,
ls.obj.3=year.to.october )
names(myList)
```

```
## [1] "ls.obj.1" "ls.obj.2" "ls.obj.3"
```

Lists

- We can use the dollar syntax to access list items (in fact, a data frame is a special type of list):

```
myList$ls.obj.1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

- We can also use `myList[[1]]` to get the first item in the list.

```
 myList[[1]]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Indexing data frames and matrices

- You can index multidimensional data structures like matrices and data frames using commas. If you don't provide an index for either rows or columns, all of the rows or columns will be returned.
  - `object[rows, colums]`

```
e[1,2]
```

```
## [1] 6
```

```
e[1,]
```

```
## [1] 1 6
```

```
patients[1,2]
```

```
## [1] "Jones"
```

Indexing data frames and matrices

```
patients[1,]
```

```
##   First_Name Second_Name  Full_Name
## 1       Adam       Jones Adam Jones
##    Sex Age Weight Consent
## 1 Male  50   70.8    TRUE
```

Advanced indexing

- As values in R are really vectors, so indices are actually vectors, and can be numeric or logical:

```
s <- letters[1:5]
s[c(1,3)]
```

```
## [1] "a" "c"
```

```
s[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
```

```
## [1] "a" "c"
```

Advanced indexing

```
a<- 1:5
a < 3
```

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

```
s[a<3]
```

```
## [1] "a" "b"
```

```
s[a > 1 & a <3]
```

```
## [1] "b"
```

```
s[a==2]
```

```
## [1] "b"
```

Operators

- arithmetic `+`, `-`, `*`, `/`, `^`
- comparison `<`, `>`, `<=`, `>=`, `==`, `!=`
- logical `!`, `&`, `|`, `xor`

Exercise

- Create a data frame called `my.patients` using the instructions in the slides. Change the data if you like.
- Check you have created the data frame correctly by loading the original version from this file in the Day_1_scripts folder using `source`

```
source("1.2_patients.R")
```

- Remake your data frame with three new variables: country, continent, and height. Make up the data. Make country a character vector but continent a factor.

Exercise

- Try the `summary` function on your data frame. What does it do? How does it treat vectors (numeric, character, logical) and factors? (What does it do for matrices?)

```
summary(my.patients)
```

- Use logical indexing to select the following patients:
- Patients under 40
- Patients who give consent to share their data
- Men who weigh as much or more than the average European male (70.8 kg)

Logical indexing answers

- Patients under 40

```
patients[patients$Age<40,]
```

- Patients who give consent to share their data

```
patients[patients$Consent==TRUE,]
```

```
##    First_Name Second_Name     Full_Name
## 1        Adam       Jones    Adam Jones
## 2         Eve      Parker    Eve Parker
## 4        Mary       Davis    Mary Davis
## 8     Matthew       Smith Matthew Smith
## 10      Sally      Wilson  Sally Wilson
##       Sex Age Weight Consent
## 1    Male  50   70.8    TRUE
## 2  Female  21   67.9    TRUE
## 4  Female  45   61.9    TRUE
```

Logical indexing answers

- Men who weigh as much or more than the average European male (70.8 kg):

```
patients[patients$Sex=="Male" & patients$Weight>=70.8,]
```

```
##   First_Name Second_Name     Full_Name
## 1       Adam        Jones    Adam Jones
## 3       John        Evans    John Evans
## 5      Peter        Baker   Peter Baker
## 8    Matthew        Smith Matthew Smith
## 9      David     Roberts David Roberts
##    Sex Age Weight Consent
## 1 Male  50   70.8    TRUE
## 3 Male  35   75.3   FALSE
## 5 Male  28   72.4   FALSE
## 8 Male  33   71.5    TRUE
## 9 Male  57   73.2   FALSE
```

R for data analysis

3 steps to Basic Data Analysis

1. Reading in data
   - `read.table()`
   - `read.csv()`, `read.delim()`
2. Analysis
   - Manipulating & reshaping the data
   - Any maths you like
   - Plotting the outcome
3. Writing out results
   - `write.table()`
   - `write.csv()`

A simple walkthrough

- 50 neuroblastoma patients were tested for NMYC gene copy number by interphase nuclei FISH
  - Amplification of NMYC correlates with worse prognosis
  - We have count data
  - Numbers of cells per patient assayed
- We need to determine which patients have amplifications
  - (i.e $>$33% of cells show NMYC amplification)

1. Read in the data

- The data is a tab-delimited file. Each row is a record, each column is a field. Columns are separated by tabs in the text
- We need to read in the results and assign it to an object (`rawdata`)

```
rawData <- read.delim("1.3_NBcountData.txt")
rawData[1:10,]
```

- If the data has been comma-separated then, `sep=","`

```
read.csv("1.3_NBcountData.csv")
```

- For full list of arguments

```
?read.table
```

1. Read in the data

```
rawData[1:10,]
```

```
##    Patient Nuclei NB_Amp NB_Nor NB_Del
## 1        1     65      0     63      2
## 2        2     51      3     43      5
## 3        3     37      2     35      0
## 4        4     37      2     35      0
## 5        5     45      2     42      1
## 6        6     46      4     41      1
## 7        7     65      1     64      0
## 8        8     59      1     54      4
## 9        9     49      0     48      1
## 10      10     46      0     45      1
```

Handling missing values

- The data frame contains some *NA* values, which means the values are missing – a common occurrence in real data collection
- `NA` is a special value that can be present in objects of any type (logical, character, numeric etc)
- `NA` is not the same as `NULL`. `NULL` is an empty R object. `NA` is one missing value within an R object (like a data frame or a vector)
- Often R functions will handle *NA*s gracefully, but sometimes we have to tell the functions what to do with them. R has some built-in functions for dealing with *NA*s, and functions often have their own arguments (like `na.rm`) for handling them

Handling missing values

```
x<-c(1,NA,3)
mean(x)
```

```
## [1] NA
```

```
mean(x,na.rm=TRUE)
```

```
## [1] 2
```

```
mean(na.omit(x))
```

```
## [1] 2
```

```
 is.na(x)
```

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

2. Analysis (reshaping data and maths)

- Our analysis involves identifying patients with $>$ 33% NB amplification

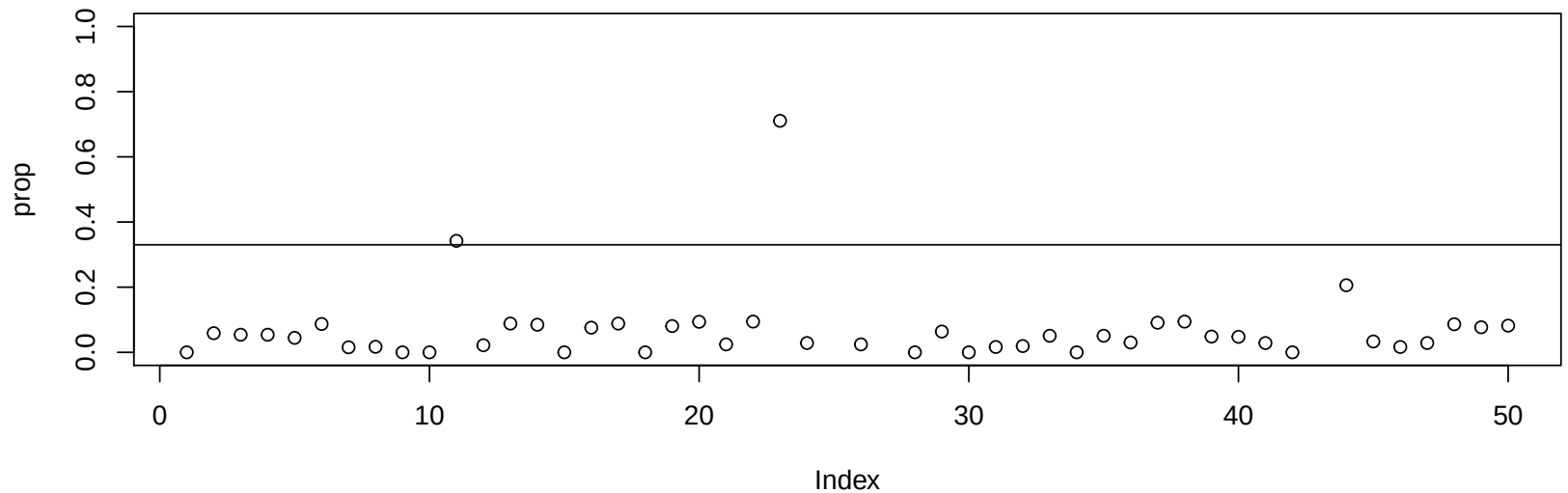
```
prop <- rawData$NB_Amp / rawData$Nuclei
# create an index of results
amp <- which(prop > 0.33)
# Get sample names of amplified patients
```

2. Analysis (reshaping data and maths)

- We can plot a simple chart of the % NB amplification

```
plot(prop, ylim=c(0,1))

abline(h=0.33)
```



3. Outputting the results

- We write out a data frame of results (patients $>$ 33% NB amplification) as a 'comma separated values' text file

```
write.csv(rawData[amp,],file="selectedSamples.csv")
```

```
+ The output file is directly-readable by Excel
```

Its often helpful to double check where the data has been saved

```
getwd() # print working directory
list.files() # list files in working directory
```

Data analysis exercise: Which samples are near normal?

- Patients are near normal if;
  - `(NB_Amp / Nuclei < 0.33 & NB_Del == 0)`
- Modify the condition in our previous code to find these patients
- Write out a results file of the samples that match these criteria, and open it in a spreadsheet program

Solution to NB normality test

```
norm <- which(prop < 0.33 & rawData$NB_Del == 0)
norm
```

```
##  [1]  3  4  7 15 20 24 36 37 42 47
```

```
write.csv(rawData[norm,], "My_NB_output.csv")
```

Programming techniques

Basic R 'built-in' functions for working with objects

- R has many built-in functions for doing simple calculations on objects. Start with a random sample of 15 numbers from 1 to 100 and try the functions below

```
x<-sample(100,15)
```

- Arithmetic with vectors
- Min / Max value number in a series

```
min(x);max(x)
```

Basic R 'built-in' functions for working with objects

- Sum of values in a series

```
sum(x)
```

- Summary statistics

```
mean(x) ; median(x)
```

- Range of values in a series

```
range(x)
```

Basic R 'built-in' functions for working with objects

- Variance / standard deviation

```
var(x);sd(x)
```

Arithmetic with vectors

- Rank ordering

```
rank(x)
```

- Quantiles

```
quantile(x)
```

Working with data frames

- We have seen before how we can get the names of our variables, but for data frames and matrices we can also get these names with colnames, and the names of the rows with rownames

```
names(patients)
```

```
## [1] "First_Name"  "Second_Name"
## [3] "Full_Name"   "Sex"
## [5] "Age"         "Weight"
## [7] "Consent"
```

```
colnames(patients)
```

```
## [1] "First_Name"  "Second_Name"
## [3] "Full_Name"   "Sex"
## [5] "Age"         "Weight"
## [7] "Consent"
```

Working with data frames

```
rownames(patients)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"
##  [8] "8"  "9"  "10"
```

Working with data frames

- We can get the numbers of rows or columns with `nrow` and `ncol`

```
nrow(patients)
```

```
## [1] 10
```

```
ncol(patients)
```

```
## [1] 7
```

```
dim(patients)
```

```
## [1] 10  7
```

Working with data frames

- We can also find the length of a vector or a list with length, although this may give confusing results for some structures, like data frames:

```
length(c(1,2,3,4,5))
```

```
## [1] 5
```

```
length(patients)
```

```
## [1] 7
```

```
length(patients$Age)
```

```
## [1] 10
```

Working with data frames

Remember, a data frame is a list of variables, so its length is the number of variables. The length of one of the variable vectors (like Age) is the number of observations

Working with data frames

- We can add rows or columns to a data frame using rbind and cbind

```
newpatient<-c("Kate","Lawson","Kate Lawson",
"Female","35","62.5","TRUE")
tail(rbind(patients,newpatient))
```

```
##    First_Name Second_Name
## 6        Paul     Daniels
## 7      Joanna     Edwards
## 8     Matthew       Smith
## 9       David     Roberts
## 10      Sally      Wilson
## 11       Kate      Lawson
##          Full_Name    Sex Age Weight
## 6    Paul Daniels   Male  31   69.9
## 7  Joanna Edwards Female  42   63.5
## 8   Matthew Smith   Male  33   71.5
```

Working with data frames

```
tail(cbind(patients,10:1))
```

```
##    First_Name Second_Name
## 5       Peter       Baker
## 6        Paul     Daniels
## 7      Joanna     Edwards
## 8     Matthew       Smith
## 9       David     Roberts
## 10      Sally      Wilson
##          Full_Name    Sex Age Weight
## 5     Peter Baker   Male  28   72.4
## 6    Paul Daniels   Male  31   69.9
## 7  Joanna Edwards Female  42   63.5
## 8   Matthew Smith   Male  33   71.5
## 9   David Roberts   Male  57   73.2
## 10   Sally Wilson Female  62   64.8
##    Consent 10:1
## 5     FALSE    6
```

Working with data frames

- We can also remove rows and columns

```
patients[-1,] # remove first row patients[,-1]
```

Working with data frames

- We can also remove rows and columns

```
patients[,-1] # remove first column
```

Working with data frames

- Sorting a vector with `sort`

```
sort(patients$Second_Name)
```

```
##  [1] "Baker"   "Daniels" "Davis"
##  [4] "Edwards" "Evans"   "Jones"
##  [7] "Parker"  "Roberts" "Smith"
## [10] "Wilson"
```

- Sorting a data frame by one variable with `order`

```
order(patients$Second_Name)
```

```
##  [1]  5  6  4  7  3  1  2  9  8 10
```

Working with data frames

```
patients[order(patients$Second_Name),]
```

```
##    First_Name Second_Name
## 5       Peter       Baker
## 6        Paul     Daniels
## 4        Mary       Davis
## 7      Joanna     Edwards
## 3        John       Evans
## 1        Adam       Jones
## 2         Eve      Parker
## 9       David     Roberts
## 8     Matthew       Smith
## 10      Sally      Wilson
##          Full_Name    Sex Age Weight
## 5     Peter Baker   Male  28   72.4
## 6    Paul Daniels   Male  31   69.9
## 4      Mary Davis Female  45   61.9
## 7  Joanna Edwards Female  42   63.5
```

The R workspace

- The objects we have been making are created in the R workspace
- When we load a package, we are loading that package's functions and data sets into our workspace
- You can see what is in your workspace with `ls`

```
ls()
```

The R workspace

- You can attach data frames to your workspace and then refer to the variables directly

```
attach(patients)
Full_Name
```

```
##  [1] "Adam Jones"     "Eve Parker"
##  [3] "John Evans"     "Mary Davis"
##  [5] "Peter Baker"    "Paul Daniels"
##  [7] "Joanna Edwards" "Matthew Smith"
##  [9] "David Roberts"  "Sally Wilson"
```

- You can remove objects from the workspace with `rm`

```
x<-1:5
rm(x)
```

The R workspace

- Your workspace is like an unsaved Word document
- When you quit R, it will usually save your workspace to a hidden file called `'.Rdata'` in your current directory. This workspace will be loaded again if you open R in the same directory
- This file is a binary, computer-readable file, not a human-readable file, which you have to open with R (like a Word document in Office)

The R workspace

- It is safer to explicitly save your workspace using `save.image`

```
save.image("phd.chapter.1.R")
```

- This way, if you are working on several different projects, you can make sure the objects for each project are saved to named files, rather than trying to remember which directory you were working in, or risking overwriting some objects you forgot about and need later.
- To load a particular image, use `load`

```
load("phd.chapter.1.R")
```

Packages in The R workspace

- You can see which packages are loaded into your workspace with `search`

```
search()
```

```
##  [1] ".GlobalEnv"
##  [2] "patients"
##  [3] "package:knitr"
##  [4] "package:stats"
##  [5] "package:graphics"
##  [6] "package:grDevices"
##  [7] "package:utils"
##  [8] "package:datasets"
##  [9] "package:methods"
## [10] "Autoloads"
## [11] "package:base"
```

- `.GlobalEnv` is where all the objects you create are stored

Packages in The R workspace

- `search` shows the search path R runs through whenever you use an object or function name. It will first look in your global environment, then in the Rstudio tools (if using Rstudio), then in the stats package and so on
- When loading packages, you will often see warnings about some objects or functions being 'masked'. This means that the newly loaded package contains an object with the same name as some object in a package that is already loaded. R will use the object in the new package whenever it comes across the name, because the new package will be earlier in the search path

Introducing loops

- Many programming languages have ways of doing the same thing many times, perhaps changing some variable each time. This is called *looping*
- Loops are not used in R so often, because we can usually achieve the same thing using vector calculations
- For example, to add two vectors together, we do not need to add each pair of elements one by one, we can just add the vectors

Introducing loops

- But there are some situations where R functions can not take vectors as input. For example, `read.csv` will only load one file at a time
- What if we had ten files to load in, all ending in the same extension (like `.csv`)

Introducing loops

- We could do this:

```
colony<-data.frame() # Start with empty data frame
colony1<-read.csv("11_CFA_Run1Counts.csv")
colony2<-read.csv("11_CFA_Run2Counts.csv")
colony3<-read.csv("11_CFA_Run3Counts.csv")
  ...
colony10<-read.csv("11_CFA_Run10Counts.csv")
colony<-rbind(colony1,colony2,colony3,...,colony10)
```

- But this will be boring to type, difficult to change, and prone to error
- As we are doing the same thing 10 times, but with a different file name each time, we can use a **loop** instead

Example `for` loop

- A `for` loop: run some code on every value in a vector

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

Example `while` loop

- `while` loop: run some code while some condition is true

```
i <- 1
while ( i <= 10 ) {
print(i)
i <- i + 1
}
```

Loops Commands and flow control

- Here's how we might use a `for` loop to load in our CSV files
- If the data files are in your current working directory, we can look up files containing a particular substring in their name using the `dir` function

```
dir(pattern="Counts.csv")
```

Loops Commands and flow control

- So we can load all the files using a `for` loop as follows

```
colony<-data.frame() countfiles<-dir(pattern="Counts.csv")
for (file in countfiles) {
   t<-read.csv(file)
   colony<-rbind(colony,t)
}
```

- Here, we use a temporary variable `t` to store the data in each file, and then add that data to the main colony data frame.

Conditional branching: Commands and flow control

- Use an `if` statement for any kind of condition testing
- Different outcomes can be selected based on a condition within brackets

```
if (condition) {
... do this ...
} else {
... do something else ...
}
```

- `condition` is any logical value, and can contain multiple conditions. e.g. `(a==2 & b <5)`, this is a compound conditional argument

Conditional branching: Commands and flow control

- For example, if we were writing a script to load an unknown set of files, using the for loop we wrote before, we might want to warn the user if we can't find any files with the pattern we are searching for. Here's how we can use an if statement to test for this

Conditional branching: Commands and flow control

```
colony<-data.frame()
countfiles<-dir(pattern="Counts.csv")
if (length(countfiles) == 0) {
    stop("No Counts.csv files found!")
} else {
    for (file in countfiles) {
        t<-read.csv(file)
        colony<-rbind(colony,t)
    }
}
```

- The stop function outputs the error message and quits

Code formatting avoids bugs!

```
f<-26
while(f!=0){
print(letters[f])
f<-f-1 }
```

Code formatting avoids bugs!

```
f <- 26
while( f != 0 ){
   print(letters[f])
f <- f-1 }
```

Code formatting avoids bugs!

- The code between brackets {} *always* is *indented*, this clearly separates what is executed once, and what is run multiple times
- Trailing bracket } always alone on the line at the same indentation level as the initial bracket {
- Use white spaces to divide the horizontal space between units of your code, e.g. around assignments, comparisons

Exercise

- Output the patients data frame, with the patients sorted in order of age, oldest first. (You may need the `rev` function)
- Load in the `colony` data frame using a `for` loop. Three of the data files are in the `Day_1_scripts` folder. Load all three files into colony using the for loop in the slides
- How many observations do you have in the colony data frame? Find out by counting the number of rows in colony using the `nrow` function
- Suppose a power analysis of your data shows that you only need 48 observations to robustly test your hypothesis. This means we can stop loading files when we have loaded at least 48 observations. Modify your for loop so it will only load files if the colony data frame has less than 48 observations in it

Answers to Exercise

- To order the patients by decreasing age

```
patients[rev(order(patients$Age)),]
```

- To find the number of rows in the colony data frame

```
nrow(colony)
```

Answers to Exercise

- To stop loading files after at least 48 observations have been found, use the code from the first `for` loop slide with a new `if` statement

```
colony<-data.frame()
countfiles<-dir(pattern="Counts.csv")
for (file in countfiles) {
    if ( nrow(colony) < 48 ) {
        t<-read.csv(file)
        colony<-rbind(colony,t)
} }
```

End of Day 1