(R)efreshing R Skills

Joseph Bonello
Faculty of ICT

UNIVERSITY OF MALTA
L-Università ta' Malta

Introduction

- Who is the tutor?
- What will we cover?
- How is this course organized?

Training Outline

- Day 1
  - Lesson 1: Introducing R
    - Install R and introduce basic features.
  - Lesson 2: Vectors and Matrices
    - Using Vectors and Matrices.
  - Lesson 3: Functions
    - Introduction to functions and basic programming principles in R.

Training Outline

- Day 2
  - Continue where we left off on previous day
  - Lesson 4: Data Frames
    - Introduction to Data Frames and functions used to load/save data from/to files.
  - Lesson 5: Manipulating Data Frames
    - Using dplyr to manipulate data frames.
  - Lesson 6: Basic Visualisation
    - Using the Base Package to visualise data

Training Outline

- Day 3
  - Continue where we left off on previous day
  - Lesson 7: Advanced Visualisations using ggplot2

Training Outline

- Sessions
  - Sessions are *hands-on* sessions
  - I will be available during sessions, so if something does not work, we'll fix it together ☺
  - Each day is divided into three sessions
    - First (long) session of around 1½ hours
    - Break (15 minutes)
    - Second session of around 1 hour
    - Break (15 minutes)
    - Final session of around 1 hour

Lesson 1

Introducing R

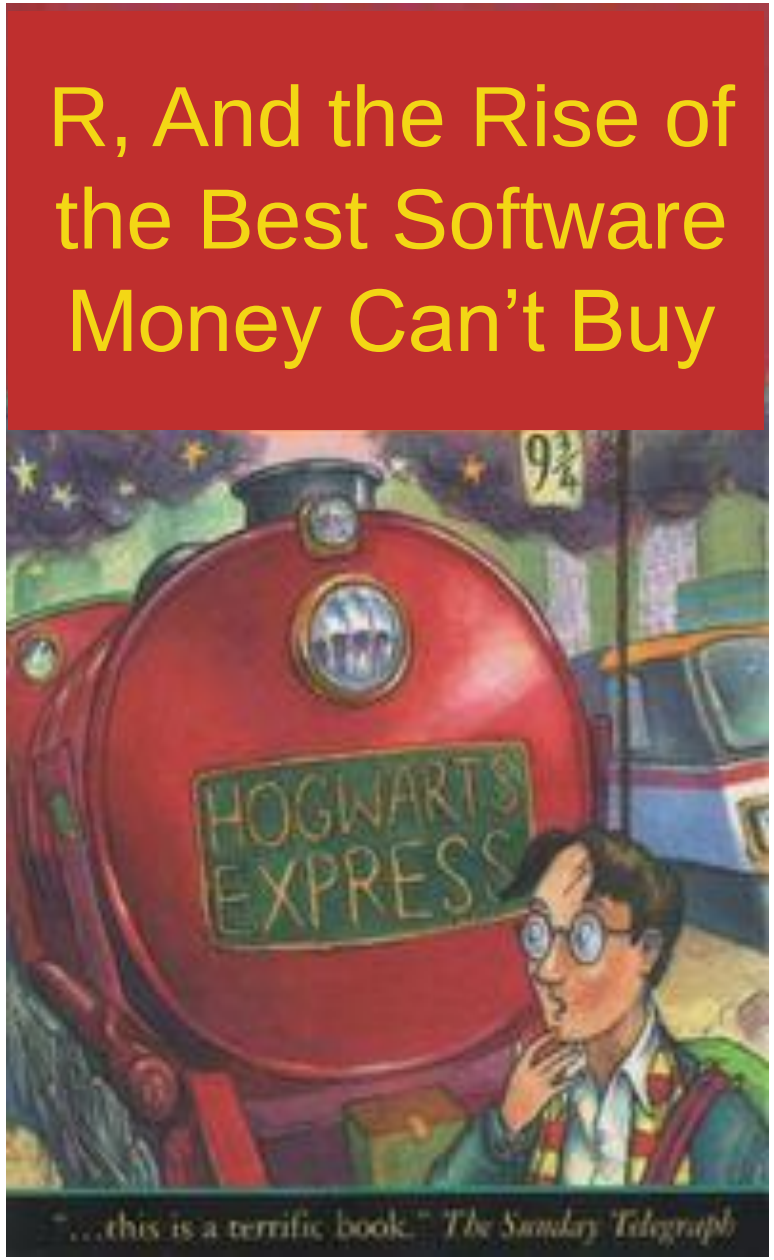
Lesson 1: Objectives

- Introduction to R
  - Brief history of R
  - Installing R (Linux, Windows)
  - Familiarisation with the tools
  - Finding help
  - R Data Types

Introduction to R

- R is an **open source** package for *statistical computing*
- Based on the language S, developed at Bell Labs (1976)
- R is designed by **Ross Ihaka** and **Robert Gentleman** at Univer[illegible] of Auckland, New Zealand in 1993
  - Current Stable Release (as at 13/08/2016: **3.3.1 released 21/06/2016**, codename Bug in Your Hair)
- R is an **interpreted**, **command driven**, **dynamic typed** language
  - It is also a **multi-paradigm** language(supported paradigms: array, object-oriented, imperative, functional, procedural, reflective)

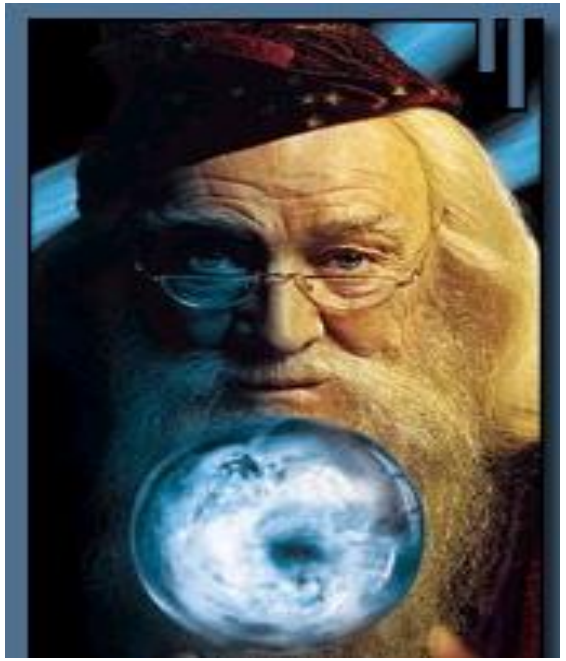
R programming language is a lot like magic... except instead of spells you have functions.



SPSS =

muggle

SPSS and SAS users are like muggles. They are limited in their ability to change their environment. They have to rely on algorithms that have been developed for them. The way they approach a problem is constrained by how SAS/SPSS employed programmers thought to approach them. And they have to pay money to use these constraining algorithms.

R =  wizard

R users are like wizards. They can rely on functions (spells) that have been developed for them by statistical researchers, but they can also create their own. They don’t have to pay for the use of them, and once experienced enough (like Dumbledore), they are almost unlimited in their ability to change their environment.

R Advantages and Disadvantages

Advantages

- **Fast** and **free**.
- **State of the art**: Statistical researchers provide their methods as R packages. SPSS and SAS are years behind R!
- **2nd** only to **MATLAB** for graphics.
- **Active** user community
- Excellent for **simulation**, **programming**, **computer intensive analyses**, etc.
- Interfaces with database storage software (SQL)

Disadvantages

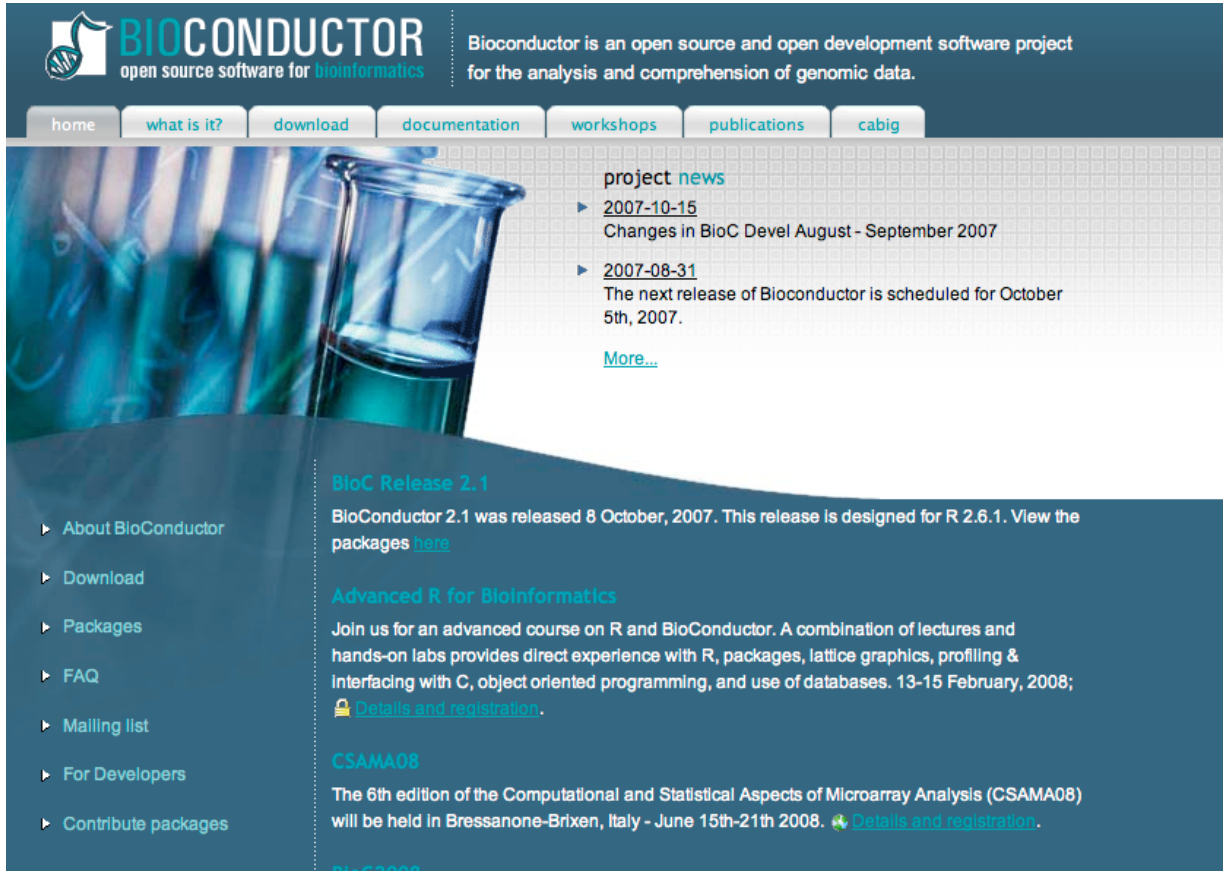
- **Not user friendly** at start - steep learning curve, minimal GUI.
- **No commercial support**; figuring out correct methods or how to use a function on your own can be frustrating.
- **Easy** to **make mistakes** and not know.
- **Data prep and cleaning** can be **messier** and more **mistake prone** in R than SPSS or SAS

So learning R can be …

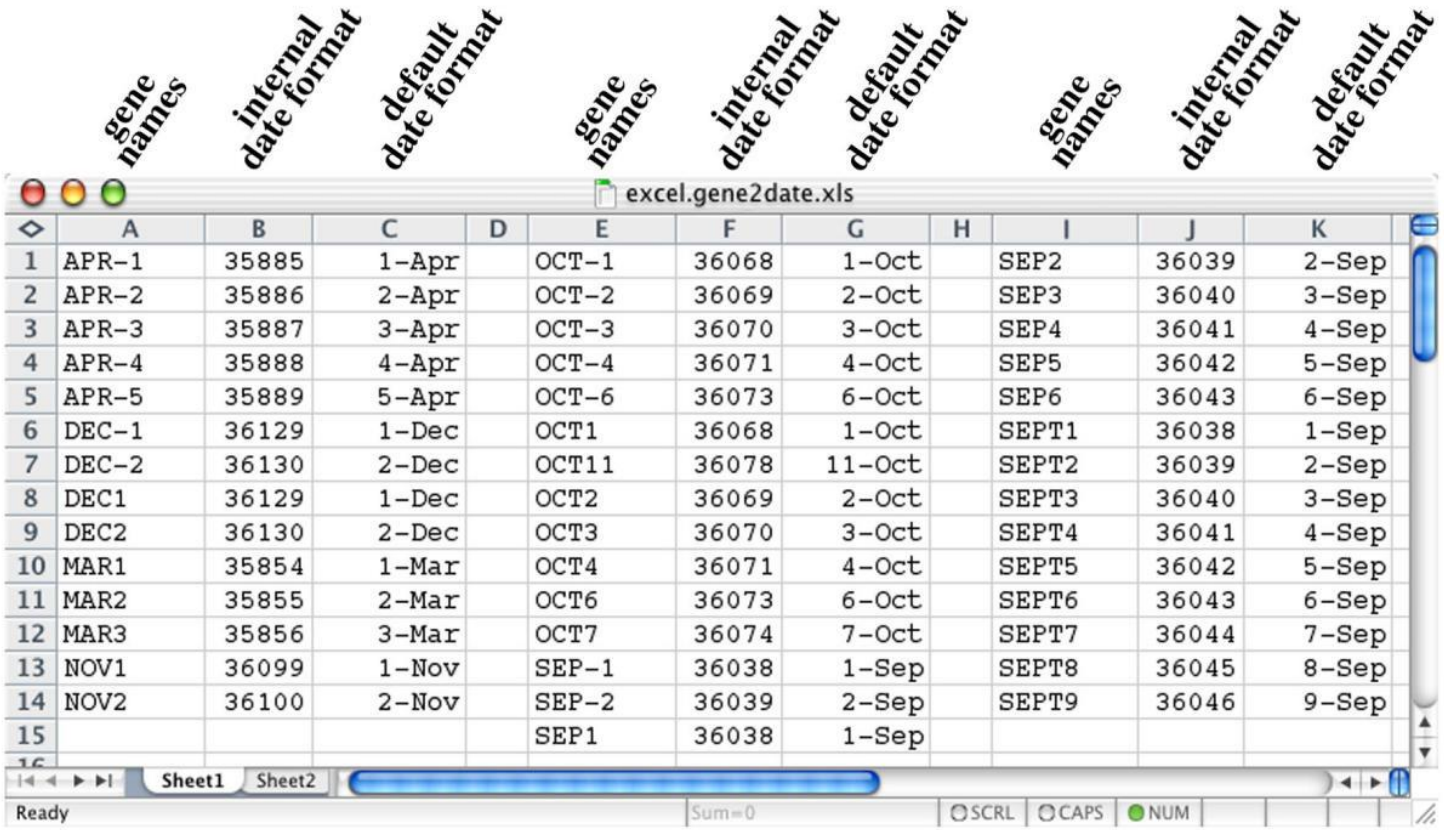
Stackoverflow will help ☺

A particular R strength: in genomics

- **Bioconductor** is a suite of **additional functions** and some **200 packages** dedicated to **analysis**, **visualization**, and **management** of genomic data



The Perils of Excel



	A	B	C	D	E	F	G	H	I	J	K
1	APR-1	35885	1-Apr		OCT-1	36068	1-Oct		SEP2	36039	2-Sep
2	APR-2	35886	2-Apr		OCT-2	36069	2-Oct		SEP3	36040	3-Sep
3	APR-3	35887	3-Apr		OCT-3	36070	3-Oct		SEP4	36041	4-Sep
4	APR-4	35888	4-Apr		OCT-4	36071	4-Oct		SEP5	36042	5-Sep
5	APR-5	35889	5-Apr		OCT-6	36073	6-Oct		SEP6	36043	6-Sep
6	DEC-1	36129	1-Dec		OCT1	36068	1-Oct		SEPT1	36038	1-Sep
7	DEC-2	36130	2-Dec		OCT11	36078	11-Oct		SEPT2	36039	2-Sep
8	DEC1	36129	1-Dec		OCT2	36069	2-Oct		SEPT3	36040	3-Sep
9	DEC2	36130	2-Dec		OCT3	36070	3-Oct		SEPT4	36041	4-Sep
10	MAR1	35854	1-Mar		OCT4	36071	4-Oct		SEPT5	36042	5-Sep
11	MAR2	35855	2-Mar		OCT6	36073	6-Oct		SEPT6	36043	6-Sep
12	MAR3	35856	3-Mar		OCT7	36074	7-Oct		SEPT7	36044	7-Sep
13	NOV1	36099	1-Nov		SEP-1	36038	1-Sep		SEPT8	36045	8-Sep
14	NOV2	36100	2-Nov		SEP-2	36039	2-Sep		SEPT9	36046	9-Sep
15					SEP1	36038	1-Sep				

Screen shot of Microsoft Excel spreadsheet illustrating errors caused by default conversion of gene names to dates. Columns A, E, and I contain the correct gene names. Columns B, F, and J contain the corresponding underlying internal Excel date representation resulting from the forced default date conversion.

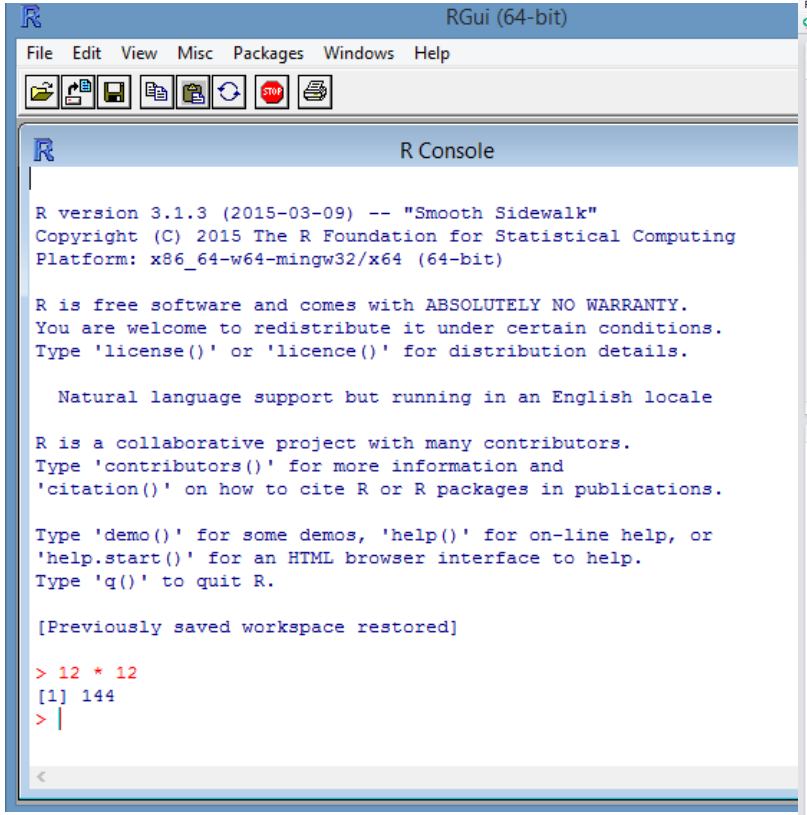
Source: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-5-80 (http://bit.ly/2bHFVuj)

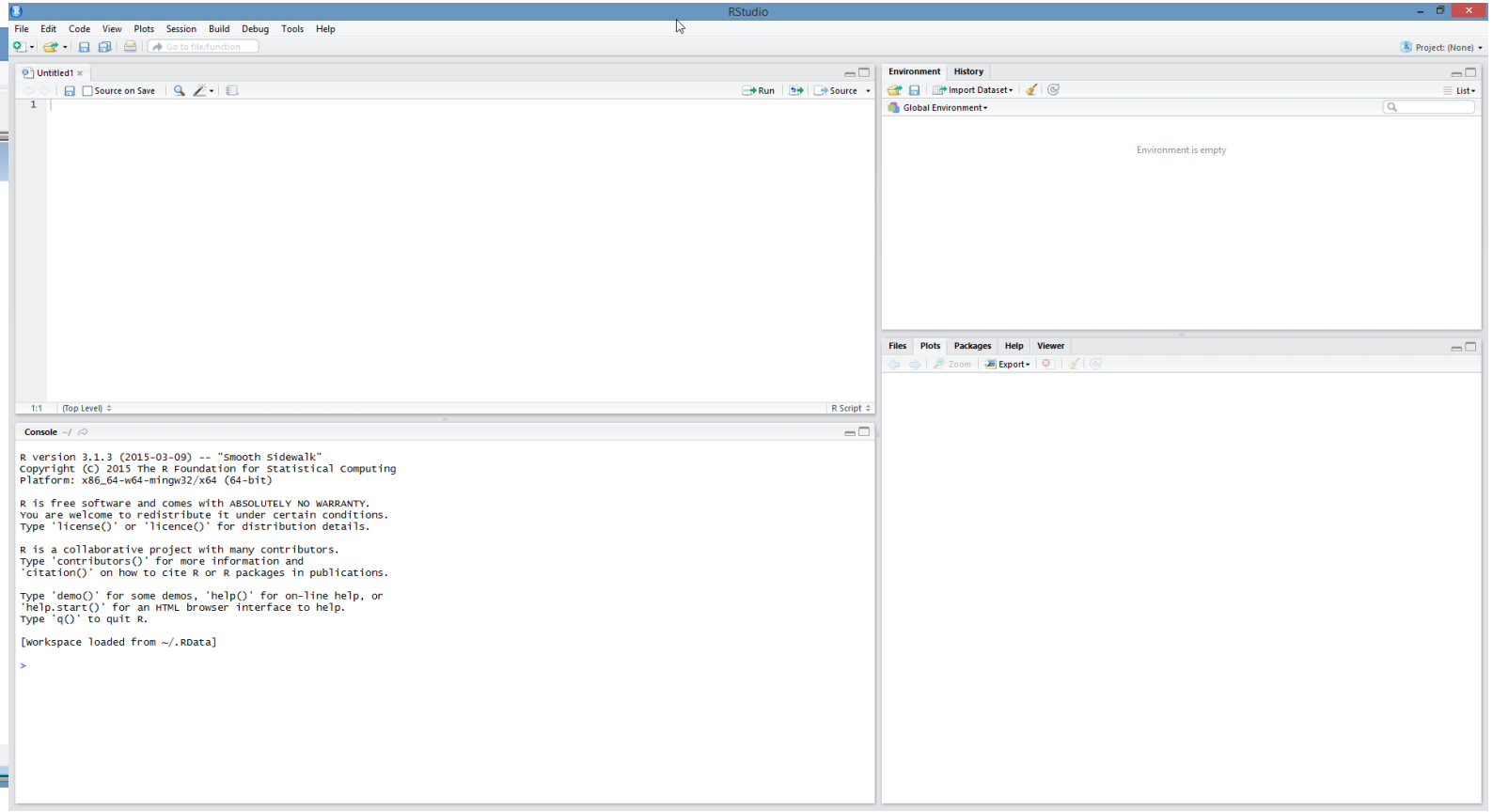
Introduction to R

- R is downloadable from http://cran.r-project.org
  - Download the Base package
- RStudio is a standard IDE for R
  - It is downloadable from https://www.rstudio.com/
- R is used for statistical computing and data analysis

Meet R

- R's interfaces





Installing R in Ubuntu - Terminal

sudo apt-get install **r-base** *r-doc-info* r-doc-pdf

Installing R on Windows and Mac

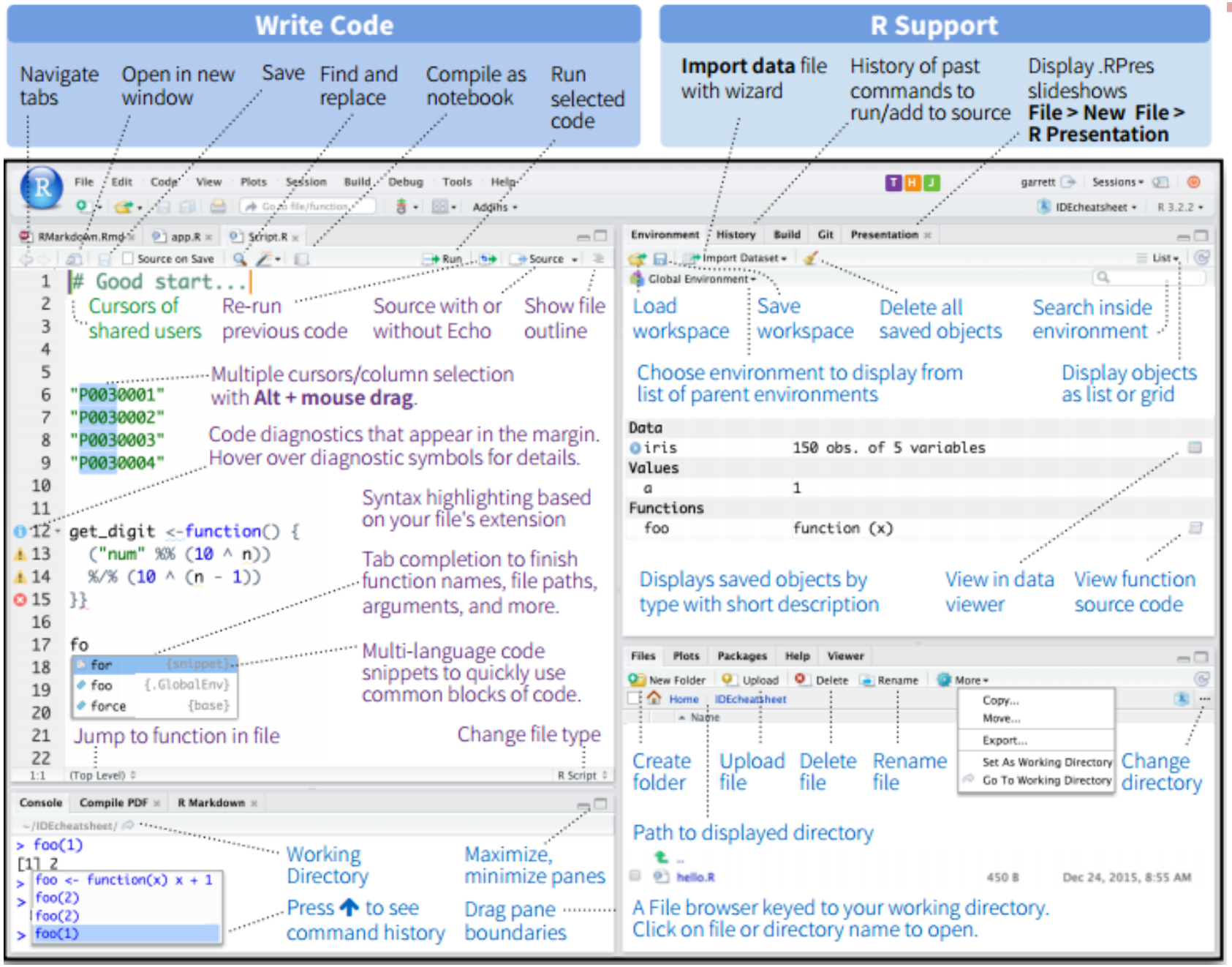
- You can download R from https://cloud.r-project.org/
- Choose Download R for your Operating System

- From the chosen mirror, download the base and contrib installers

Installing R in Ubuntu –Software Centre

Download Rstudio from: **http://ow.ly/XPXFo**

RStudio


Write Code
Navigate tabs
Open in new window
Save
Find and replace
Compile as notebook
Run selected code
R Support
Import data file with wizard
History of past commands to run/add to source
Display .RPres slideshows File > New File > R Presentation
Cursors of shared users
Re-run previous code
Source with or without Echo
Show file outline
Multiple cursors/column selection with Alt + mouse drag.
Code diagnostics that appear in the margin. Hover over diagnostic symbols for details.
Syntax highlighting based on your file's extension
Tab completion to finish function names, file paths, arguments, and more.
Multi-language code snippets to quickly use common blocks of code.
Jump to function in file
Change file type
Load workspace
Save workspace
Delete all saved objects
Search inside environment
Choose environment to display from list of parent environments
Display objects as list or grid
Displays saved objects by type with short description
View in data viewer
View function source code
Create folder
Upload file
Delete file
Rename file
Change directory
Path to displayed directory
A File browser keyed to your working directory. Click on file or directory name to open.
Working Directory
Maximize, minimize panes
Press ↑ to see command history
Drag pane boundaries

A word of caution

- PowerPoint has a tendency to “beautify” certain characters
  - Much to the *despair* of who wants to copy/paste code into an R editor
  - One problematic character is the quote (")
  - This is what R expects "A and B"
  - This is what PowerPoint gives you “A and B”
  - In R, this is what happens

```
> print(“A and B”)
Error: unexpected input in "print(”"
```

  - Be careful when copy/pasting!

A word of caution

- PowerPoint has a tendency to “beautify” paragraphs!

- Some functions may get wrapped on multiple lines ….

qplot(vore, **fill**=order, **data**=msleep,
geom="bar")

- Anything between parenthesis “()” goes on the same line

qplot(vore, **fill**=order, **data**=msleep, **geom="bar"**)

Language Introduction

R Command Line

- R allows commands to be executed at the command line
- At the prompt (typically denoted by > sign), type

```
12 * 12
```

- On pressing enter, R evaluates the expression and returns

```
[1] 144
```

as the answer

- Note: The *[1]* indicates the result has only one line of output. Results may have more than one line and R formats the output to aid the users.

R – Getting Help

- Two ways of getting help
  - Using the help() function
    - help(sum)
    - help.search("histogram")
    - ?hist
  - Using the example() function
    - example(rep)
  - Remember that R is case sensitive!

Comments in R

- Comments in R are preceded by the # symbol

```
# I am a comment in R! All characters are ignored
# 12 * 12
# Nothing happens; it's a comment
```

Mathematical operators in R

Operator	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Power	^
Modulus	%%
Integer Division	%/%
Greater Than, Greater Than or Equal To	>, >=
Less Than, Less Than or Equal To	<, <=
Equality, Non-equality	==, !=
Logical NOT, AND, OR	!, &&, ||

R – Practical 1: Using R as a calculator

- Use the console on the right to try out examples of the operators used in R at the prompt

Operator	Code
Addition	5 + 7
Subtraction	11 – 5
Multiplication	8 * 9
Division	45 / 4
Power	2 ^ 5
Modulus	45 %% 4
Integer Division	45 %/% 4

R - Variables

- R has a *global environment* which stores:
  - The **results** of calculations
  - Other **types** of objects
- Variables in R are **created upon assignment**
- Variables are **assigned** using the '**<-**' symbol
- For example:

Variable Name

r.cm <- 15 Value

*circle.area.cm <- pi * (r.cm^ 2)*

print(circle.area.cm)

R – Practical 1.2: Using Variables

- Calculate the perimeter of a square, 25 X 25 cm
- Calculate the perimeter of a rectangle, 25 X 50 cm
- Output the value after each operation
- Create a variable and initialise it to 8 (print output after each operation)
  - Add 2 to it.
  - Subtract 4 from it
  - Multiply it by 5
  - Divide it by 3
  - Increment the variable by one
  - Decrement the variable by one

```
r.cm <- 15
circle.area.cm <- pi * (r.cm^ 2)
print(circle.area.cm)
```

R – Data Types

Data Type	Sample(s)
Numeric	v <- NA w <- NaN x <- 1e+07 y <- 125.544 z <- -5.3
Integer	w <- as.integer(TRUE) x <- as.integer(178) y <- as.integer(1e+07) z <- as.integer(5.3) a <- as.integer(‘Joseph’)

R – Data Types

Data Type	Sample(s)
Character	x <- “one” y <- ‘two’ z <- “three”
Logical	w <- T x <- F y <- TRUE z <- FALSE
Complex	x <- 1+1i

R – Practical 1.3: Data Types

```
# Numeric Data Types
v <- NA
w <- NaN
x <- 1e+07
y <- 125.544
z <- -5.3

print(v)
print(w)
print(x)
print(y)
print(z)
```

Output value of v

R – Practical 1.3: Data Types

```
# Numeric Data Types
v <- NA
w <- NaN
x <- 1e+07
y <- 125.544
z <- -5.3

print(v == v) # NA == NA
print(v == w) # NA == NaN
print(5 == v) # 5 == NA
print(8 == w) # 8 == NaN
```

R – Practical 1.3 : Data Types

```
# Integer Data Types
w <- as.integer(TRUE)
x <- as.integer(178)
y <- as.integer(1e+07)
z <- as.integer(5.3)

print(w)
print(x)
print(y)
print(z)

a <- as.integer('Joseph')
print(a)
```

R – Practical 1.3 : Data Types

```
# Character data Types
x <- "one“
y <- 'two‘
z <- 't‘

print(x)
print(y)
print(z)
```

R – Practical 1.3 : Data Types

```
# Logical Data Types
w <- T
x <- F
y <- TRUE
z <- FALSE

print(w)
print(x)
print(y)
print(z)
```

R – Practical 1.3 : Data Types

```
# Complex Data Types
x <- 1+1i
print(x)
```

R – Data Structures

Data Type	Description	Sample(s)
Vectors	A vector is **a sequence of data elements** of the **same basic type**. Members in a vector are officially called components.	x <- c(1,2,3,4) y <- c(T, F, TRUE, FALSE) z <- c("My", "name", "is","Joseph") for(i in seq_along(z)) print(z[i])

Sequences	**Sequences** return a **vector** within the given range.	v <- seq(3,12)
		w <- 1:3
		x <- seq(0,25, by=5)
		y <- seq_len(5) # create seq up to 5
		z <- seq_along(c(2,5,7,9)) # create a sequence up to length of vector

Matrices	A matrix is a **collection of data elements** arranged in a **two-dimensional rectangular layout**.	x <- matrix(c(1,2,3,4,5,6), nrow=2, ncol = 3, byrow=T)

R – Data Structures

Data Type	Description	Sample(s)
Arrays	A **collection** of data **elements** arranged in a **multi-dimensional** array. The **second argument** of the array represents the **dimensions** of the array.	a <- array(1:24, c(3,4,2))
Factors	A **factor** is a vector that can contain only **predefined values**, and is **used to store categorical data**. Factors are built on top of integer vectors using two attributes: the class(), “factor”, which makes them behave differently from regular integer vectors, and the levels(), which defines the set of allowed values.	x <- factor(c("a", "b", "b", "a")) gender_char <- c("m", "m", "m”, ”f”) gender_factor <- factor(gender_char, levels = c("m", "f"))
Lists	Lists are **ordered sequences** of objects which can be of **any mode**: the first object of the list may be a vector, the second a matrix, the third another list, etc.	x <- list(name="Joseph", age=18, likesReading=T) print(x$name)

R – Practical 1.4: Data Structures

```
# Vectors
x <- c(1,2,3,4)
y <- c(T, F, TRUE, FALSE)
z <- c("My", "name", "is","Joseph")

print(x)
print(y)
print(z)
```

R – Practical 1.4: Data Structures

```
# Sequences
v <- seq(3,12)
w <- 1:3
x <- seq(0,25, by=5)
y <- seq_len(8)  # create seq up to 8
z <- seq_along(c(2,5,7,9)) # create a sequence up to length of vector

print(v)
print(w)
print(x)
print(y)
print(z)
```

R – Practical 1.4: Data Structures

```
# Matrices
x <- matrix(c(1,2,3,4,5,6),nrow=2,ncol = 3,byrow=T)
print(paste('By Row',x))
```

paste concatenates strings

```
y <- matrix(c(1,2,3,4,5,6),nrow=2,ncol = 3,byrow=F)
print(paste('By Column', y))
```

R – Practical 1.4: Data Structures

```
# Factors
x <- factor(c("a", "b", "b", "a"))
print(class(x))
```

Variable type: Factor

```
print(levels(x))
```

Returns the set of allowed values

```
gender_char <- c("m", "m", "m", "f")
gender_factor <- factor(gender_char, levels = c("m", "f"))
print(gender_char)
print(gender_factor)
```

R – Practical 1.4: Data Structures

```
# Arrays
a <- array(1:24, c(3,4,2))
print(a)

# Lists
x <- list(name="Joseph", age=18, likesReading=T)
print(x$name)
```

Lesson 1: Wrap-up

- In this session ….
  - Installed R
  - Introduced RStudio
  - Used R as a calculator
  - Introduced R variables
  - Introduced R operators
  - Introduced R datatypes

Lesson 2

Vectors and Matrices

Lesson 2: Objectives

- Working with R
  - Useful commands
  - Two fundamental data structures
    - Vectors
    - Matrices

Getting the version of R

- R.Version()

```
> R.Version()
$platform
[1] "x86_64-pc-linux-gnu"

$arch
[1] "x86_64"

$os
[1] "linux-gnu"

$system
[1] "x86_64, linux-gnu"

$status
[1] ""

$major
[1] "3"

$minor
[1] "3.1"

$year
[1] "2016"

$month
[1] "06"

$day
[1] "21"

$`svn rev`
[1] "70800"
```

Useful R Functions

- Determining the class of an object
  - **x <- 1**
  - **class(x)**
- Read first few rows
  - **head(1:1000, 15)**
- Display/Save/Load commands history
  - **history(max.show = 25, reverse = TRUE)**
  - **savehistory(file = “sessionHistory”)**
  - **loadhistory(file = “sessionHistory”)**

Useful R Functions

- Installing and using a library in R
  - **install.packages("ggplot2")**
  - **library(ggplot2)**

  - **Note:** Many functions in R live in **optional** packages. The **library()** function
    - lists packages,
    - shows help, or
    - loads packages from the package library.

- Quitting R
  - **q()**

Vector Arithmetic

- Suppose we are using:
  - **a <- c(1, 3, 5, 7)**
  - **b <- c(1, 2, 4, 8)**
- Simple Arithmetic
  - **a + b**
  - **a – b**
  - **5 * a**
  - **a / b**
  - **sum(a)**
  - **mean(a)**

Vector Arithmetic

- **Recycling** vectors
  - This happens when two vectors are of **unequal** length
  - The **shorter** one will be **recycled** to **match** the longer vector
    - Example
      - **u <- c(10, 20, 30)**
      - **v <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)**
      - **u + v**

Combining Vectors

- Vectors can be **combined** using the c function
- Example
  - **n <- c(2, 3, 5)**
  - **s <- c("aa", "bb", "cc", "dd", "ee")**
  - **c(n, s)**

- What do you note about the result?

Vector Operations

- Accessing Indices
  - **s <- c("aa", "bb", "cc", "dd", "ee")**
  - **s[3]**
  - **s[2:4] # Range index**
  - **s[c(2, 1, 3)] # Out of Order Indexes**
  - **s[c(2,3,3)] # Duplicate Indexes**

Vector Operations

- Strip/Remove a member from the resulting vector (not deleted!)
  - **s[-3]**

- Out or range vectors give NA
  - **s[10]**

- Using elements of a vector
  - **s[1] + s[5]**

Vector Operations

- Named Vector Members
  - **people <- c(“Albert“, “Mary”, “Thomas”,”Roberta”)**
  - **names(people) <- c(“Father”, “Mother”,”Son”,”Daughter”)**
  - **people[“Daughter”]**

R – Operations on Matrices

- Generating numbers according to some distribution
  - Normally distributed matrix
    - **x <- matrix(rnorm(20), 4)**
  - Gamma distributed matrix
    - **y <- matrix(dgamma(5e-4,0.001),4)**
  - Poisson distributed matrix
    - **z <- matrix(ppois(16, lambda=12),200)**

R – Practical 2.1 : Matrices

```
x <- matrix(rnorm(20, mean=15), 4)
print(x)
```

Generate 20 normally distributed values

```
y <- matrix(pgamma(5e-4,0.001),4)
print(y)
```

Generate random deviates from the Gamma distribution

```
z <- matrix(ppois(16, lambda=12),200)
print(z)
```

Generate Poisson distributed values

R – Operations on Matrices

- Appending a vector to a matrix: cbind and rbind
  - **v1 <- c(1, 1, 2, 2)**
  - **mtx1 <- cbind(x, v1)** ← Append to a column
  - **print (mtx1)**

  - **v2 <- c(1:6)**
  - **mtx2 <- rbind(mtx1, v2)** ← Append to a row
  - **print (mtx2)**

R – Operations on Matrices

- Determine the dimensions of a matrix
  - **dim(mat7)**

- Matrix Arithmetic
  - **mtx3 <- matrix(1:6, 2)**
  - **mtx4 <- matrix(c(rep(1, 3), rep(2, 3)), 2, byrow = T)**
  - **print(mtx3 + mtx4)**
  - **print(mtx3 + 6)**
  - **print(mtx4 - mtx3)**

R – Operations on Matrices

- Determine the transpose of a matrix
  - **t(mtx3)**
- Further arithmetic
  - **print(mtx3 * mtx4)**
  - **print(mtx3 / mtx4)**
  - **print(mtx3 * 3)**
- Submatrices
  - **mtx3[, 1:2] # get the first two columns**
  - **mtx3[1,] # get the first two rows**
  - **mtx3[, 1:2]/mtx3[, 2:3]**

R – Operations on Matrices

- Give names to rows and columns
  - **dimnames(mtx3) <- list(**
    c("row1", "row2"), # row names
    c("col1", "col2", "col3")) # column names
  - **print(mtx3)**

- Reading the dimensions of an array
  - **dim(mtx3)**

Lesson 2: Wrap-up

- In this lesson:
  - We used some new R functions (head, class, history, etc.)
  - Worked with Vectors
  - Worked with Matrices

Lesson 3

Functions and Basic Programming

Lesson 3: Objectives

- Functions in R
- Flow Control
  - Conditionals
  - Iteration
- Using the apply family of functions
- Introducing some new R (base) Functions

Functions in R

- A function in R is defined as follows:

```
addTwoNumbers <- function(x, y)
{
        return(x + y)
}

toThePower <- function(x, y)
{
        return(x^y)
}
```

Functions in R

```
"
This is a multi-line comment (really a string)
The guesser function prints 'I got it'! when it guesses the hidden number (input)
using random values
"
guesser <- function( hidden )
{
        g <- -1
        attempts <- 1
        repeat {
                g <- sample(1:100,1)
                if (g == hidden) break # escape!
                else attempts <- attempts + 1
        }
        cat("I got it! Your # was", g, ". I guessed after ",attempts," attempts")
}
```

Note: Repeats Forever!

Note: double =

Function in R

- Function in R are **pass-by-value**
- R has **copy-on-modify** semantics
- Example

```
mod_vector <- function(v) {
   v[2] <- 0
   return(v)
 }

a <- c(1:10)
b <- mod_vector(a)
print(a)
print(b)
```

Flow Control: If-Else

- If statements operate on length-one logical vectors

```
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

Flow Control: If-Else

- Example:

```
v <- if(1==0) {
    print(1)
 } else {
    print(2)
}
```

Exercise: Compare

```
if(x > 1) {
        print("x is big")
} else if(x > 0) {
        print("x is positive")
} else
        print("x is negative or zero")
```

```
if(x > 1) {
        print("x is big")
}
if(x > 0) {
        print("x is positive")
}
print("x is negative or zero")
```

Flow Control: Ifelse

- Operate on vectors of variable length

- ifelse(test, true_value, false_value)

- Example

```
x <- 1:10
ifelse(x<5 | x>8, x, 0)
```

Iteration in R: The For Loop

- The syntax is: **for(*var* in *seq*) {code}**
- The *seq* determines what values *var* will take in the loop
  - The loop is **performed length(*seq*)** times
  - On the n'th iteration of the loop, *var* takes the value *seq*[n]
  - *var* is a completely new variable and **not directly related** to anything other variable

- Setting up your loop requires determining the correct *seq* to loop over: *usually easy*
- The real challenge of looping is **relating** the values of *seq* to the **dimensions/indices** of your data

Iteration in R: The For Loop

- Example:

```
index <- 1:20

squared <- rep(0, length(index))
print(squared)

listofNums <- as.integer(rpois(20, 1:100))
print(listofNums)

for(i in 1:length(index)) {
  squared[i] <- listofNums[i] ^ 2
}

print(squared)
```

Iteration in R: The For Loop

- Example 2:

```
L <- list() # empty list
L[[1]] <- 1:4
L[[2]] <- 2:7
L[[3]] <- c("a","b","c")
L[[4]] <- matrix(rnorm(4), nrow = 2)
```

Iteration in R: The For Loop

- Example 2:

```
L <- list() # empty list
L[[1]] <- 1:4
L[[2]] <- 2:7
L[[3]] <- c("a","b","c")
L[[4]] <- matrix(rnorm(4), nrow = 2)

View(L[[2]]) # View item at index = 2
```

Iteration in R: The For Loop

- Checking what L is

```
str(L)
```

- Looping on the first element of L

```
for (i in 1:length(L[[1]]))
{
  print(L[[1]][i])
}
```

Iteration in R: The For Loop

- Looping all elements

```
for (i in seq_along(L))
{
  print(paste("---------", i,"---------", sep = " ")) # paste concatenates string
  for (j in 1:length(L[[i]]))
  {
    print(L[[i]][j])
  }
}
```

Iteration in R: The For Loop

- The keyword **next** is used to **skip** an iteration

```
x <- 5
for (i in 1:100) {
        if (i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        x <- x * i
}
```

Iteration in R: The While Loop

- While loops begin by **testing a condition**
- **If** it is **true**, then they **execute** the loop body
- Once the loop body is executed, the **condition is tested again**, and so forth
- Syntax:
  - **while(condition)** statements

Iteration in R: The While Loop

- Example

```
count <- 0
while(count < 10) {
        print(count)
        count <- count + 1
}
```

- While loops can potentially result in **infinite loops** if not written properly. *Use with care!*

Iteration in R: The Repeat Loop

- Repeat initiates an **infinite loop**; these are **not commonly** used in statistical applications but may be useful
- The only way to exit the repeat loop is to use **break**
- Syntax
  - **repeat** statements

Iteration in R: The Repeat Loop

- Example

```
x <- 0
repeat
{

        if(x > 100)
        {
                break
        }
        else
        {
                x <- x + 5
        }
}
```

apply, tapply, lapply, sapply functions

- The apply family is used to **return a vector** that is applied to the margins of an array or matrix
- The function **apply** is used for **two-dimensional datasets**
- Syntax:
  - **apply**(X, MARGIN, FUN, ARGs)

where
X: array, matrix or data.frame;
MARGIN: 1 for rows, 2 for columns, c(1,2) for both;
FUN: one or more functions;
ARGs: possible arguments for function

apply, tapply, lapply, sapply functions

- Example

```
x <- 1:10
 test <- function(x) { # Defines some custom function
   if(x < 5) {
      x-1
   } else {
      x / x
   }
 }
 apply(as.array(x), 1, test)
```

apply, tapply, lapply, sapply functions

- The function **tapply** is used to apply a function to array categories of variable lengths (ragged array)
- Grouping is defined by factor
- The **tapply** function is useful when we need to
  - **break up** a vector into groups defined by some **classifying factor**,
  - **compute** a function on the subsets, and
  - return the results in a convenient form
- Note: It is possible to **specify multiple factors** as the **grouping variable**

apply, tapply, lapply, sapply functions

- Syntax:
  - **tapply**(vector, factor, FUN)

where

vector is the summary variable

factor is the group variable

FUN is the function to apply

apply, tapply, lapply, sapply functions

- Example

```
medical.example <-
        data.frame(patient = 1:100,
                   age = as.integer(rnorm(100, mean = 60, sd = 12)),
                   treatment = gl(2, 50,
                          labels = c("Treatment", "Control")))

tapply(medical.example$age, medical.example$treatment, mean)
```

Notes:

gl generates factors by specifying the pattern of their levels

mean computes the statistical mean

apply, tapply, lapply, sapply functions

- The functions lapply and sapply **apply a function** to **vector or list**
- The function **lapply** returns a list
- The function **sapply** attempts to return the simplest data object, such as vector or matrix instead of list

- Syntax
  - lapply(X, FUN)
  - sapply(X, FUN)

apply, tapply, lapply, sapply functions

- Example

```
family1 <- list(name="Jones",numofchild=2,ages=c(5,7),measles=c("Y","N"))
lapply(family1, typeof)
str(lapply(family1, typeof))

sapply(family1, function(x) {if (is.numeric(x)) mean(x)})
```

A few new tricks …

- Getting values that match some criterion
  - s <- **rnorm**(n = 1000,mean = 75,sd = 2)
  - s[s<72]

- Replace **even** elements with “z”
  - g <- c(400:600)
  - g[c(T,F)] <- "z“

- To run a source file
  - **source**("test.R")

A few new tricks …

- To find out where you are working
  - **getwd()**
- To change directory
  - **setwd**("D:/RData")
  - **setwd**("/home/joseph/RData")
- Show and describe objects in the workspace
  - **ls**()
  - **ls.str**()
- Information about the R session
  - **sessionInfo()**

A few new tricks …

- To show R's options
  - **options()**
- Save/Load all objects to a file
  - **save.image**(file="session.RData")
  - **load**("session.RData")
- Remove all objects in the workspace
  - **rm**(list=ls())
- Remove an individual item from the workspace
  - **rm**(x,y)

A few new tricks …

- To get a list of built-in datasets in R
  - **data()**
- Checking what an object consists of
  - **str**(x)
- Checking the (internal) type of an object
  - **typeof**(x)
- Concatenate and print
  - **cat**("This is number ", 9)
  - **paste**("This", "also works")
- Take a sample of the specified size from the elements of a vector
  - **sample**(1:100, 5)

Lesson 3: Wrap-up

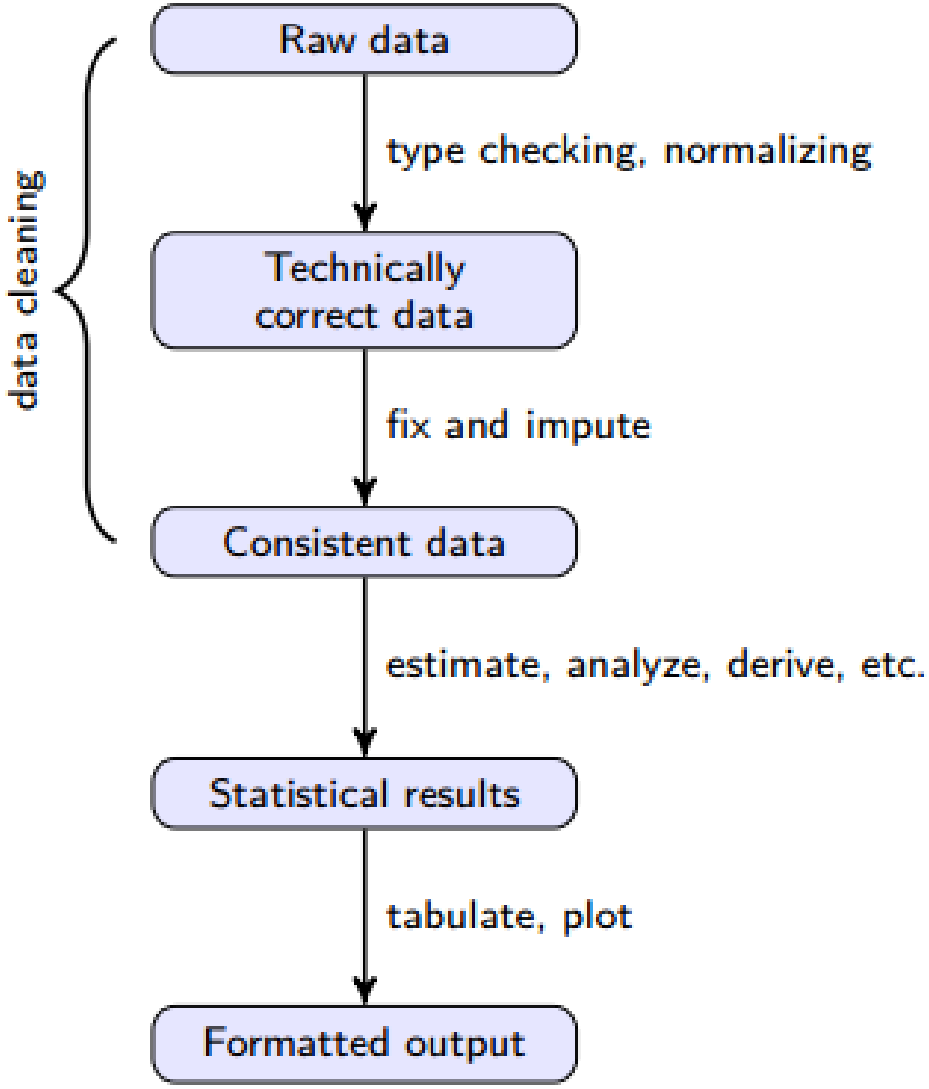
- In this lesson, we saw how to:
  - Declare *functions* in R
  - Flow Control
    - Conditionals
    - Iteration
  - Using the apply family of functions
  - Use new R Functions

Lesson 4

The Data Frame

Lesson 4: Objectives

- Using the R Data Frame
  - Introduction to the Data Frame
  - Manually creating a data frame
  - Common functions for using a data frame
  - Loading data into a data frame and retrieving data from a data frame
  - Analysing data in a data frame



Statistical Analysis Value Chain

- Technically Correct Data – Data that can be read into a *data.frame*
- Remove *inconsistencies* from data (e.g. negative age/salary, check for missing data, etc.)
- Statistical theories use consistent data as a starting point
  - Note: some data cleaning methods such as missing values imputation influence statistical results!

The Data Frame

- Data frames are used to **store tabular data**
- They are represented as a **special type of list** where every element of the list has to have the **same length**
- Each element of the list can be thought of as a **column** and the **length** of each element of the list is the **number of rows**
- Unlike matrices, data frames **can store different classes** of objects in each column (just like lists);
  - Recall that matrices must have every element be the same class

The Data Frame

- Data frames have a special attribute called **row.names**
- Data frames are usually created by calling **read.table()** or **read.csv()**
- Can be converted to a **matrix** by calling **data.matrix()**
- Constructing a data frame (the **manual** way….)

```
n <- c(2, 3, 5)
s <- c("aa", "bb", "cc")
b <- c(TRUE, FALSE, TRUE)
df <- data.frame(n, s, b)
```

Using datasets

- For this lecture, we will use the *Forbes2000* dataset
- Installing the dataset
  - **install.packages**("HSAUR")
  - **library**(HSAUR)
- Load the dataset into a data frame
  - df <- Forbes2000
- To view the data frame
  - **View**(df)

First steps in data analysis

- First steps in analysing data – take a peek at the data
  - **head**(df)
  - **tail**(df)
- Explore the dimensions of the data
  - **names**(df)
  - **rownames**(df) # may not always be applicable
  - **colnames**(df)

- Load another dataset and compare the above
  - df2 <- USArrests

First steps in data analysis

- Exploring the size of the dataset
  - **dim**(df)
  - **nrow**(df)
  - **ncol**(df)
- Explore the structure of the dataset
  - **str**(df)

Using the data

- For data frames you need **two** indices – **rows** and **columns**
  - df[1,] # First row
  - df[,2] # Second column

Note the position of the comma ‘,’ in these examples

- Reading the sales data (**column**)
  - sales <- df$sales
  - sales2 <- df[,"sales"]

Get all rows in the Sales column

- Getting the IBM data (**row**)
  - ibm_data <- df[df$name == "IBM",]

Get all columns for row IBM

Using the data

- Getting data of UK companies
  - uk_list <- df$country == “United Kingdom”
  - uk_data <- df[uk_list,]
  - uk_data_sales <- df[uk_list,]$sales
  - uk_data_sales <- df[uk_list,”sales”]

- Order the data by a particular column
  - df3 <- df[**order**(df$sales, decreasing = T),]
  - top10 <- df3[1:10,]

Some (less geeky) functions to subset data

- The **subset** function allows the selection of rows that satisfy some condition
  - subframe <- subset(df, sales > 100)

- The **which** function outputs the row numbers that match a particular condition
  - which(df$country == “China”)
  - df[which(df$country == “China”),]

- Exercise
  - Get the sales data of Korean and Japanese companies

The with() function

- The with() temporarily sets up a data frame as the default place to look up variables
- It applies an expression to a dataset

- Example:

```
richCompanies <- with(subset(df, sales > 100), {name})
print(richCompanies)
```

Preliminary Analysis

- Finding the average value of a numeric column
  - **mean**(df$sales, na.rm=T) # na.rm will remove missing values
- Finding the median value of a numeric column
  - **median**(df$sales)
- Finding the minimum of a numeric column
  - **min**(df$sales)
- Finding the maximum of a numeric column
  - **max**(df$sales)
- R provides a quick useful function to summarise data
  - **summary**(df)

Reading from files in R

- You can read from files in R using read.table or read.csv
- Example
  - df10 <- **read.csv**(‘http://goo.gl/TK8vSK’, header = F)
    - **alternatively**
  - df10 <- **read.table**(‘http://goo.gl/TK8vSK’, header = F, sep=“,”)

  - View(df10)
- Forgetting header = … in read.table() is bad (try it!)

Writing to file in R

- You can write to files in R using write.csv
- Example
  - write.csv(df10, file = “df10.csv”)

- Files, using this convention, are written to the working directory
  - You can specify the path
  - Find out **where** the file is written … we have covered the function in lesson 3

Importing from other packages

```
library("foreign")
stata <- read.dta("salary.dta")
spss <- read.spss("salary.sav", to.data.frame=TRUE)
```

- Notes:
  - The **foreign** package is in the standard distribution. It handles import and export of data.
  - Thousands of extra packages are available at http://cran.r-project.org.

Exercise (taken from Ken Rice's R for Large Data & Bioinformatics course)

1. The file http://goo.gl/bTHMIz has information on blood pressure and related variables.
   1. Read in the file, and
   2. Summarize the variables

2. The file http://goo.gl/KNQ7OD has the same data as above.
   1. Download and read it in, and
   2. Check that it is the same data as in Question 1

Source: http://faculty.washington.edu/kenrice/bigr

Lesson 4: Wrap-up

- Used the R Data Frame
  - Introduced to the Data Frame
  - Manually created a data frame
  - Used common functions for using a data frame
  - Loaded data into a data frame and retrieved data from a data frame
  - Analysed data in a data frame

Lesson 5

dplyr package

Lesson 5: Objectives

- Use R to download files locally
- Use the %in% operator
- Use the %>% operator
- Using the dplyr package
  - Select
  - Filter
  - Arrange
  - Mutate
  - Summarise
  - Group By

About dplyr

- dplyr is a powerful R package to
  - **Transform** and **summarise** tabular data
  - Used to simplify data frame manipulation

- **Why** use dplyr?
  - Working with data requires:
    - **Understanding** what needs to be done
    - **Describe** the tasks as a program
    - **Execute** it!

How does dplyr help?

- **Simplifies** common data manipulation tasks
- Provides simple functions that **correspond** to these tasks
  - **Translate** your thoughts to code!
- **Efficient**

Getting started!

- **Install** dplyr
  - install.packages("dplyr")
- **Load** dplyr
  - library(dplyr)

Dataset

- The dataset for this session is the **mammals sleep (msleep)** dataset
  - The data set contains **sleep times** and **weights** for a set of **mammals**
  - V. M. Savage and G. B. West. A quantitative, theoretical framework for understanding mammalian sleep. Proceedings of the National Academy of Sciences, 104 (3):1051-1056, 2007.

  - A convenient way to download files is to use the **downloader** package!

    - install.packages("downloader")
    - library(downloader)

Download files with dplyr

```
url <- “http://goo.gl/zebt7h”

filename <- “msleep.csv”

if (!file.exists(filename)) download(url, filename)

msleep <- read.csv(“msleep.csv”)

head(msleep)
```

dplyr – selecting the columns you need

- The dplyr verb is: **select**
- select chooses columns from the dataset

- **Syntax**
  - select(*dataset*, *column1*, *column2*, …)

- Example

  - mammal.sleep.total <- select(msleep, name, sleep_total)
  - View(mammal.sleep.total)

Selecting what is not needed

- The **subtraction (-)** operator can be used to **remove** columns that are not needed
- This is called **negative indexing**

- Example
  - all.less.name <- select(msleep, -name)

Selecting a range of columns

- To select a range of columns, use the **colon** (:) operator

- Example:
  - msleep.range <- select(msleep, name:order)
  - View(msleep.range)

Selecting columns according to a criterion

- To select columns that **start with** “sleep”, use the *starts_with* verb
  - msleep.sleep.columns <- select(msleep, starts_with(“sleep”))
  - View(msleep.sleep.columns)

- Similar verbs include:
  - ends_with(): select columns that **end with** the character string
  - contains(): select columns that **contain** the character string
  - matches(): select columns that **match a regular expression**
  - one_of(): select columns that are **from within a group** of names

Filtering rows

- The filer() function returns **rows** that **match** some condition
- The syntax is:
  - filter(*dataset*, *condition*)

- Example
  - sleepy.mammals <- filter(msleep, sleep_total >= 16)
  - View(sleepy.mammals)

Filtering rows with multiple criteria

- You can add filtering criteria by adding them in a **comma-separated list** in the filter function

- Example
  - heavy.sleepers <- filter(msleep, sleep_total >= 16, bodywt >= 1)
  - View(heavy.sleepers)

Filter rows that match a list of values

- The **%in%** operator is used to **match** values in a **column** with values in a **provided list**
- Syntax
  - filter(*dataset, columnname* **%in%** *vectorOfValues*)

- Example
  - carni.herbi.sleepers <- filter(msleep, vore %in% c(“herbi”, “carni”))
  - View(carni.herbi.sleepers)

  - lightweight.sleepers <- filter(msleep, *round(bodywt*) %in% 1:10)
  - View(lightweight.sleepers)

The pipe operator %>%

- This operator resides in another package (**magrittr**) but dplyr imports it automatically
- Used to *pipe* the **output** of one function **into** another
- Used **instead** of nesting function (read function *inside to outside*)
  - Read function *left to right* instead
  - More **natural** way of reading

- Example
  - total.sleep <- select(msleep, name, sleep_total)
  - View(total.sleep)
- Can be written as
  - msleep %>% select(name, sleep_total) %>% View

Ordering rows

- dplyr uses the **arrange** function to **sort** values in a column
- You can **order by** various columns by adding columns separated by commas
- To sort in **descending** order use the **desc** function

- Syntax
  - **arrange**(*dataset*, *columnName1*, ***desc***(*columnName2*), ...)

- Example
  - (ordered.sleep <- msleep %>% arrange(vore, desc(sleep_total))) %>% View

Create new rows

- You can create **new columns** with dplyr using the **mutate()** function
- You can create **additional columns** by separating them with commas

- Syntax
  - ***mutate***(*dataset, newColumnName1 = expression,* newColumnName2 = *expression*)

- Example
  - (new.msleep <- msleep %>% mutate(rem_proportion = sleep_rem / sleep_total)) %>% View

Summarising Data with dplyr

- The summarise() function will create **summary statistics** for a given column
- **Additional columns** can be created by separating them with commas

- Syntax
  - **summarise**(*dataset*, *newColumnName1 = fn(column1)*, *newColumnName2 = fn(column2)*, …)

- Example
  - (summarised.sleep <- msleep %>% summarise(avg_sleep = mean(sleep_total), min_sleep = min(sleep_total), **total = n()**)) %>% View

Grouping data

- This function allows the dataset to be **split** by some variable, **apply** some function on the **resulting subset** and finally **recombine** the result
- Additional columns can be added, separated by commas

- Syntax
  - **group_by**(*column1*, *column2*, …)

- Example
  - (sleep.groupedby.vore <- msleep %>% group_by(vore) %>% summarise(avg_sleep = mean(sleep_total), **total=n()**)) %>% View

Lesson 5: Wrap-up

- Used R to download files locally
- Used the %in% operator
- Used the %>% operator
- Used the dplyr package
  - Select
  - Filter
  - Arrange
  - Mutate
  - Summarise
  - Group By

Lesson 6

Basic Plots

Lesson 6: Objectives

- Use R Base package for visualisations
  - Scatter plots
  - Box plots
  - Time series
  - Histograms
  - Pie Charts
  - Barcharts

  - Simple tweaking

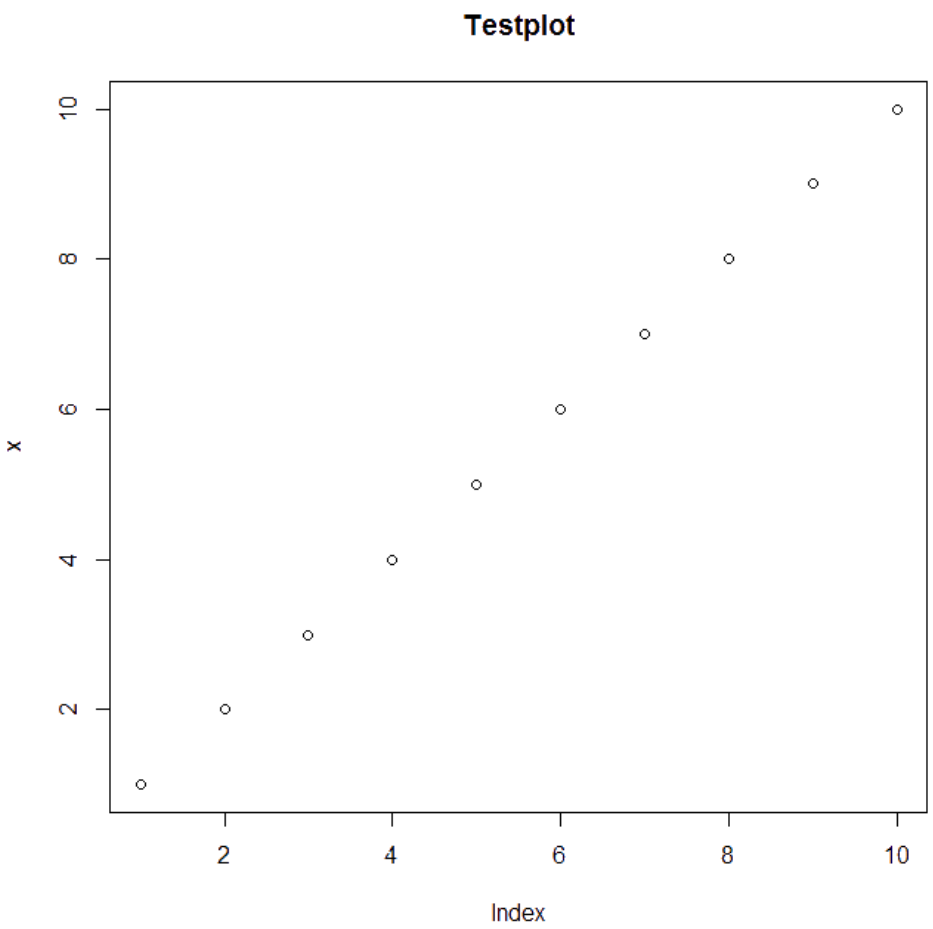
  - Save Visualisations to file

Quickly plot a vector

s <- 1:10

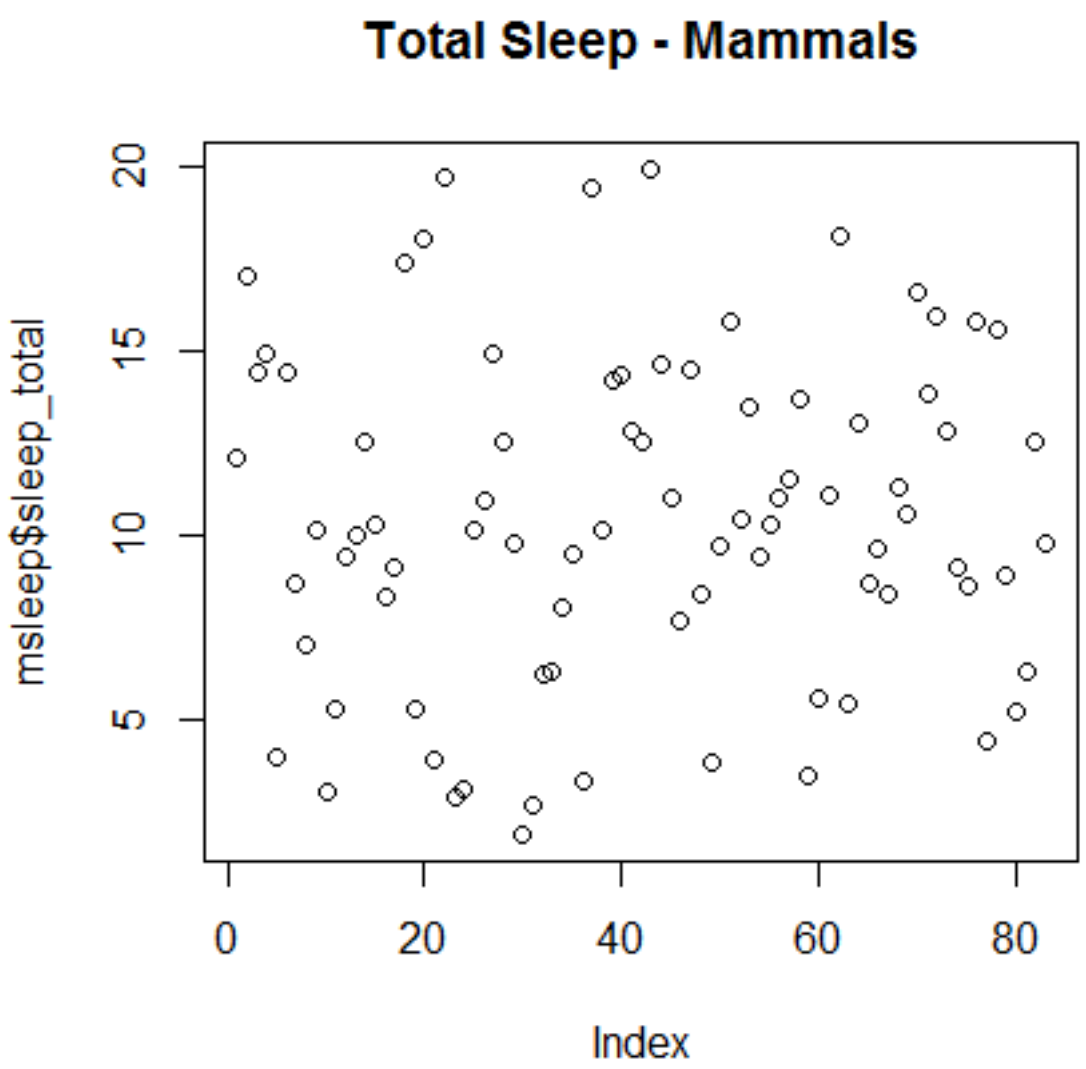
plot(s, main="Testplot")

- **main** defines a title
- **Index** (on the x-axis of the graph) shows the *index* of the vector



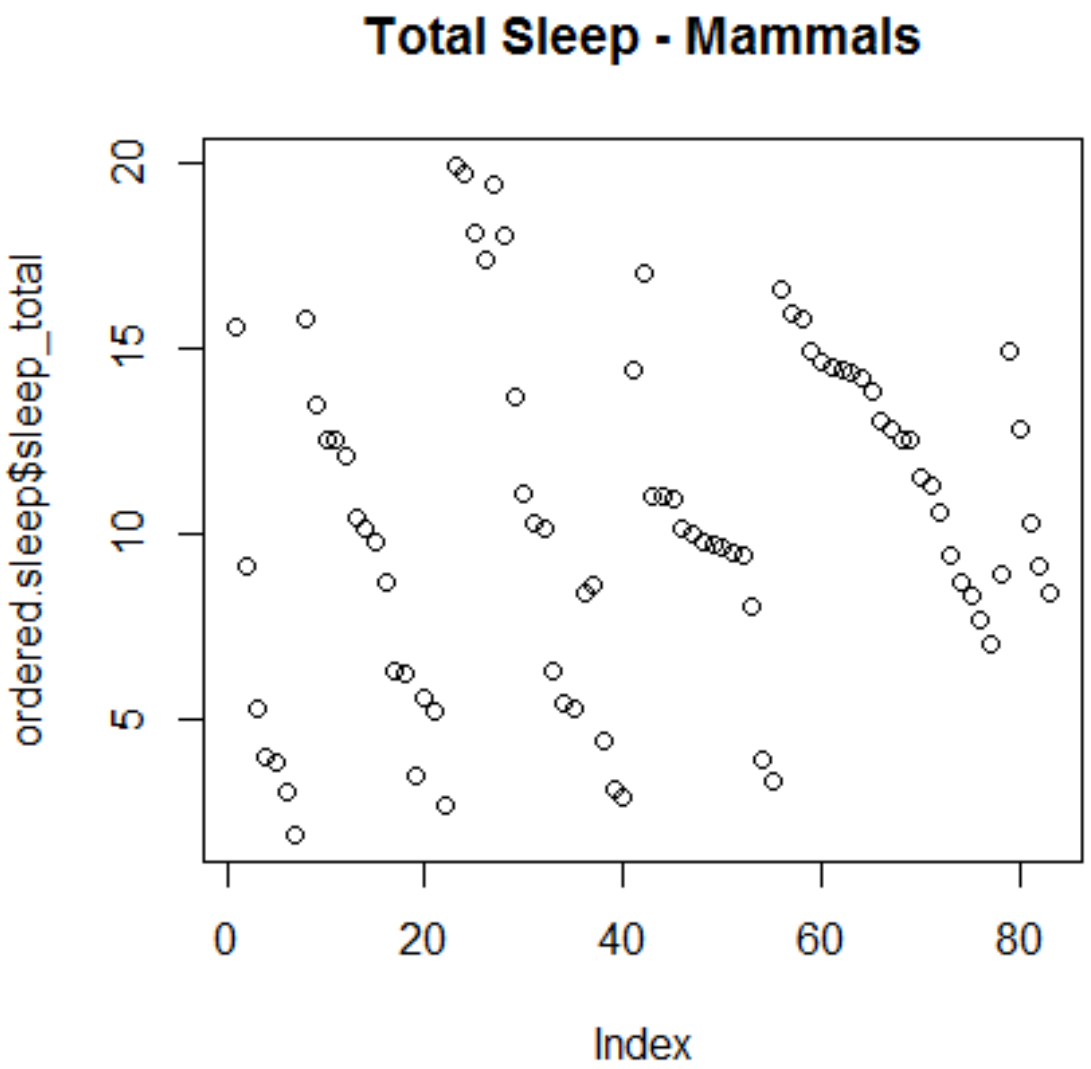
Another example

```
plot(msleep$sleep_total,
        main="Total Sleep - Mammals")
```



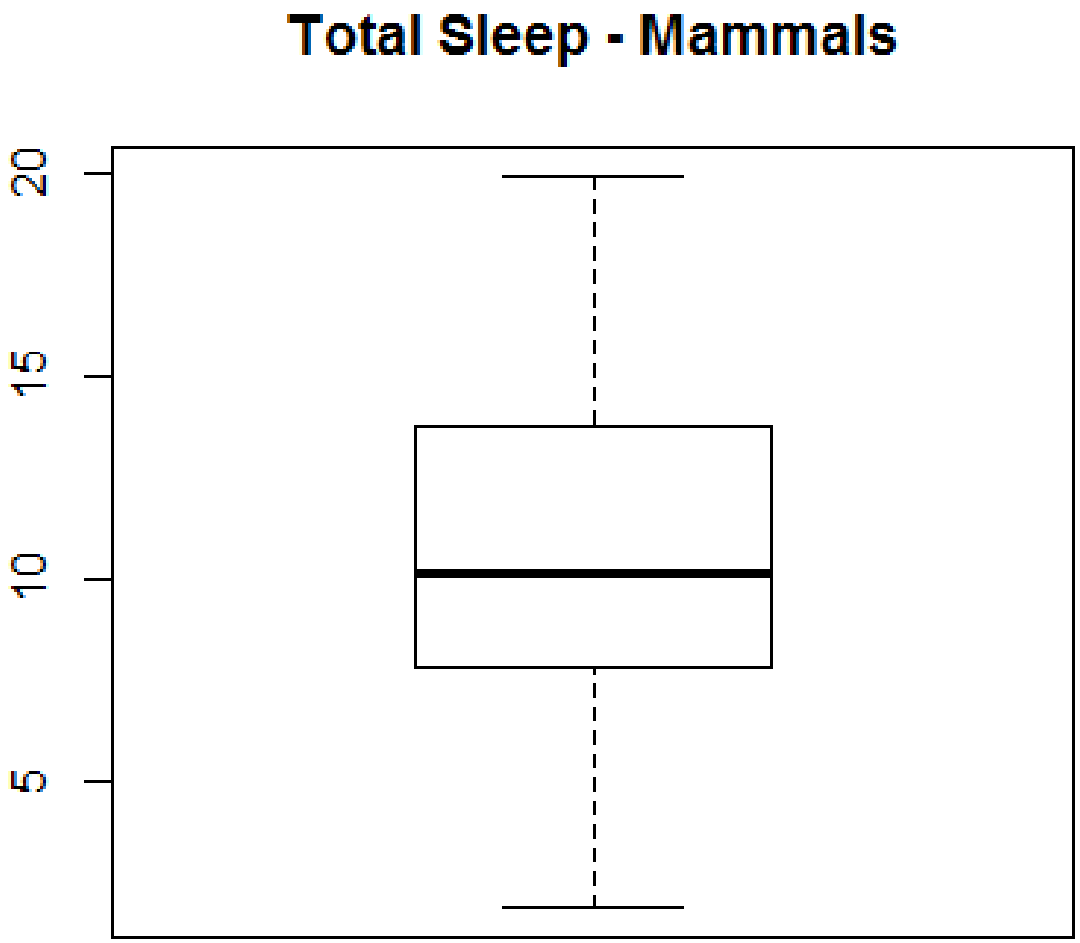
Yet Another example

```
plot(ordered.sleep$sleep_total,
        main="Total Sleep – Mammals”)
```



A box plot example

```
boxplot(msleep$sleep_total,
        main="Total Sleep - Mammals")
```



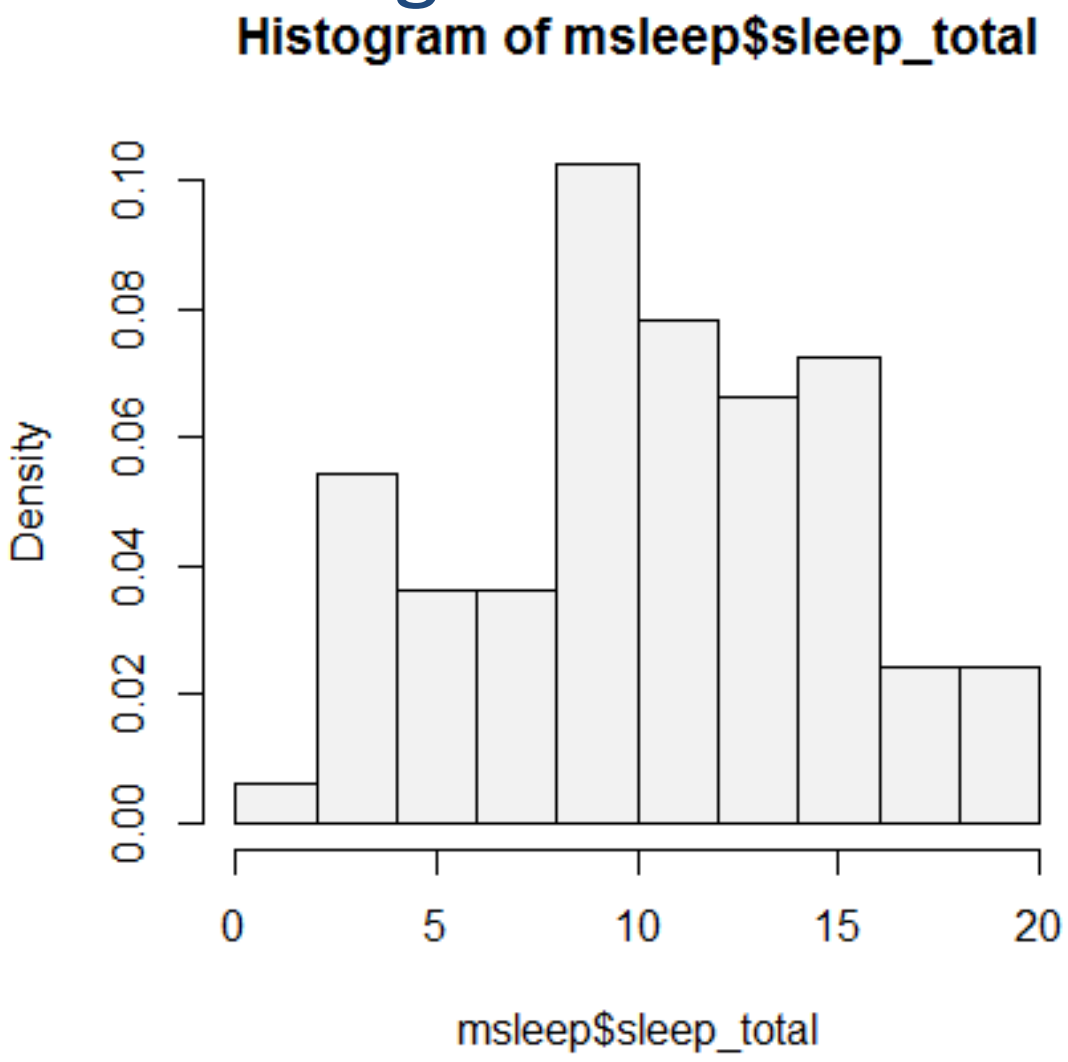
Examining distribution of data with a histogram

- Plotting Frequency

hist(msleep$sleep_total, col=gray(0.95))

- Plotting Probability Density

hist(msleep$sleep_total, col=gray(0.95), **prob=T**)



Examining distribution of data with a histogram

- Getting information about the histogram

print(**hist**(msleep$sleep_total))

```
> print(hist(msleep$sleep_total))
$breaks
 [1]  0  2  4  6  8 10 12 14 16 18 20

$counts
 [1]  1  9  6  6 17 13 11 12  4  4

$density
 [1] 0.006024096 0.054216867 0.036144578 0.036144578
 [5] 0.102409639 0.078313253 0.066265060 0.072289157
 [9] 0.024096386 0.024096386

$mids
 [1]  1  3  5  7  9 11 13 15 17 19

$xname
[1] "msleep$sleep_total"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"
```

Examining distribution of data with a histogram

- Managing the number of bins

hist(msleep$sleep_total, col=gray(0.95), **breaks=seq(0,20,1)**, prob=T, main = "Histogram of mammals total sleep")

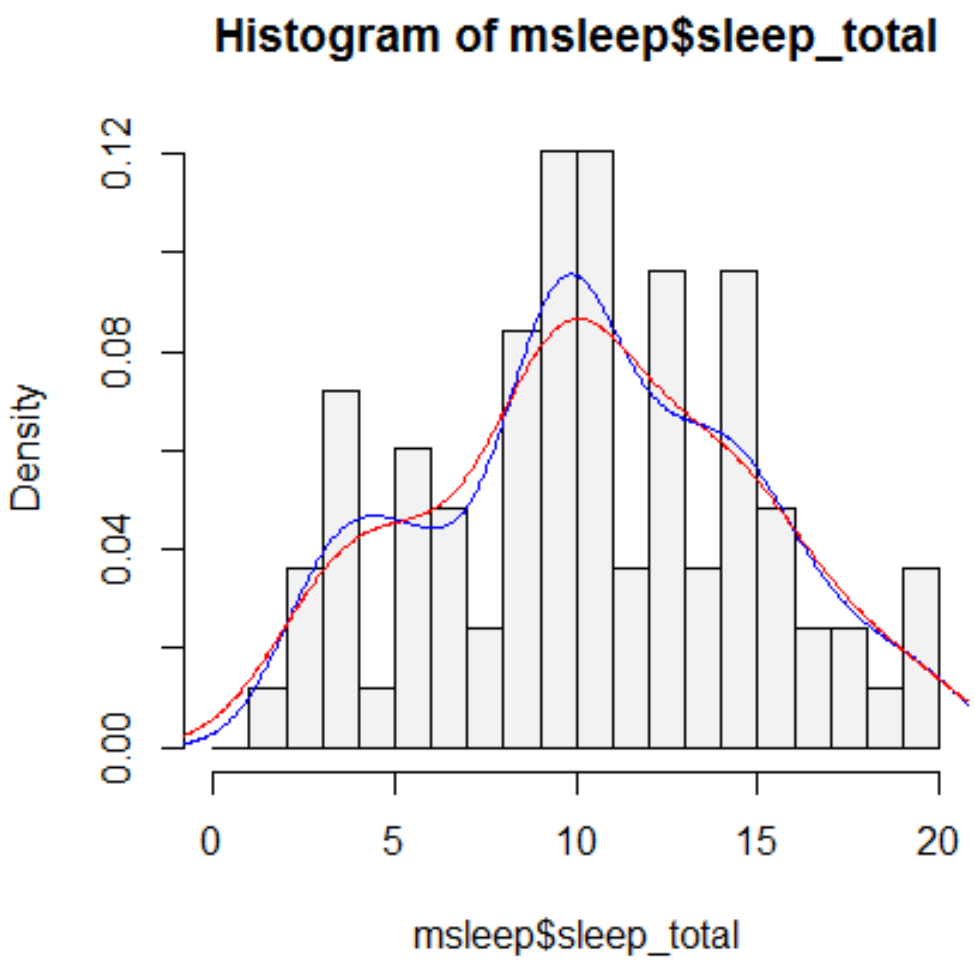
- Plotting density

lines(**density**(msleep$sleep, bw=1.302))

- Calculating bw

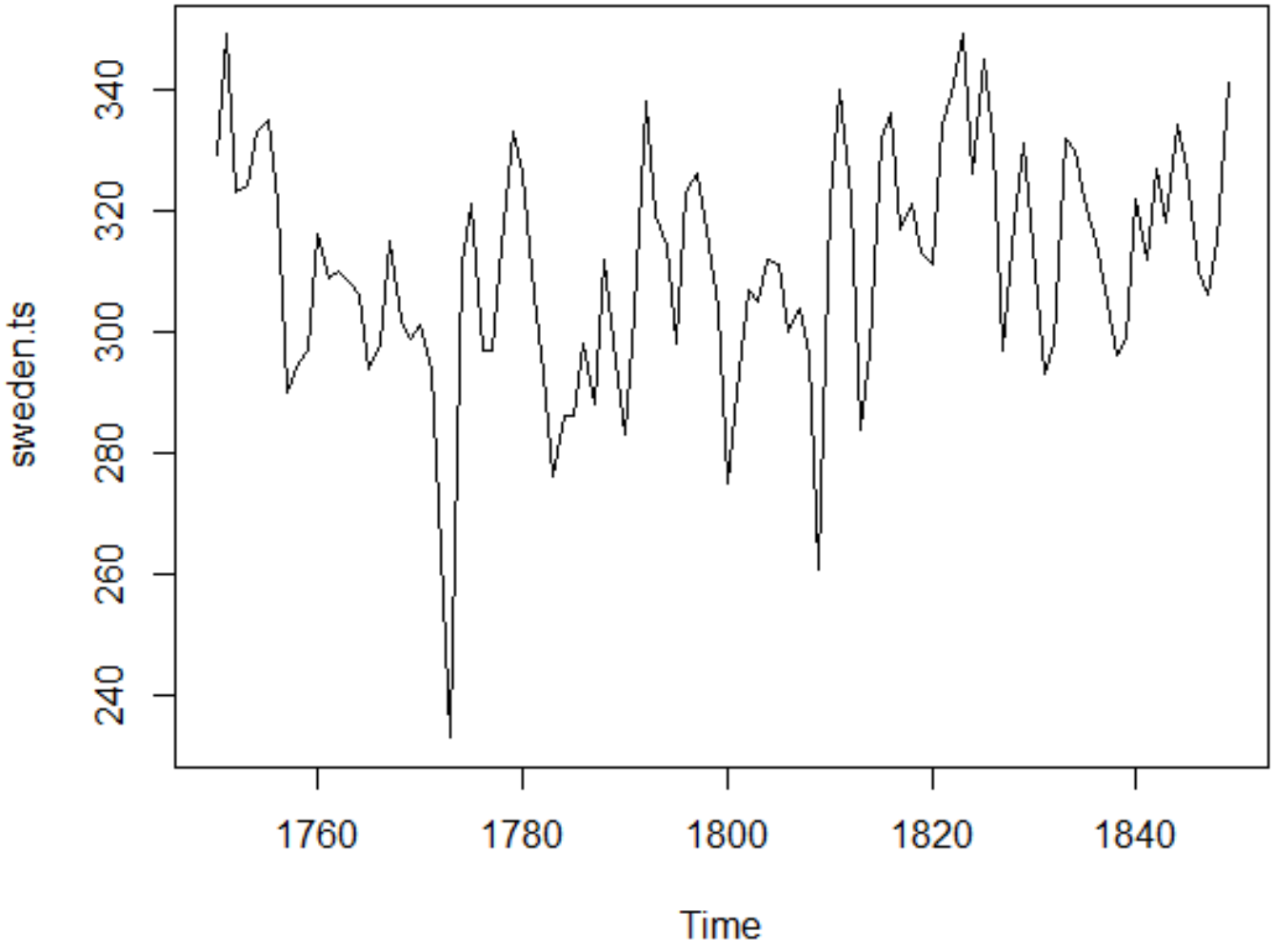
R default: **bw.nrd0**(msleep$sleep_total)

Sheather-Jones: **bw.SJ**(msleep$sleep_total, nb=**21**)



Time Series Example

```
url <- "http://goo.gl/Ob4SnZ"
filename <- "sweden.csv“

if (!file.exists(filename)) download(url, filename)

sweden <- read.csv("sweden.csv")
sweden.ts <- ts(sweden$Annual.Swedish.fertility.rates, frequency = 1, start=1750)

plot.ts(sweden.ts)
```



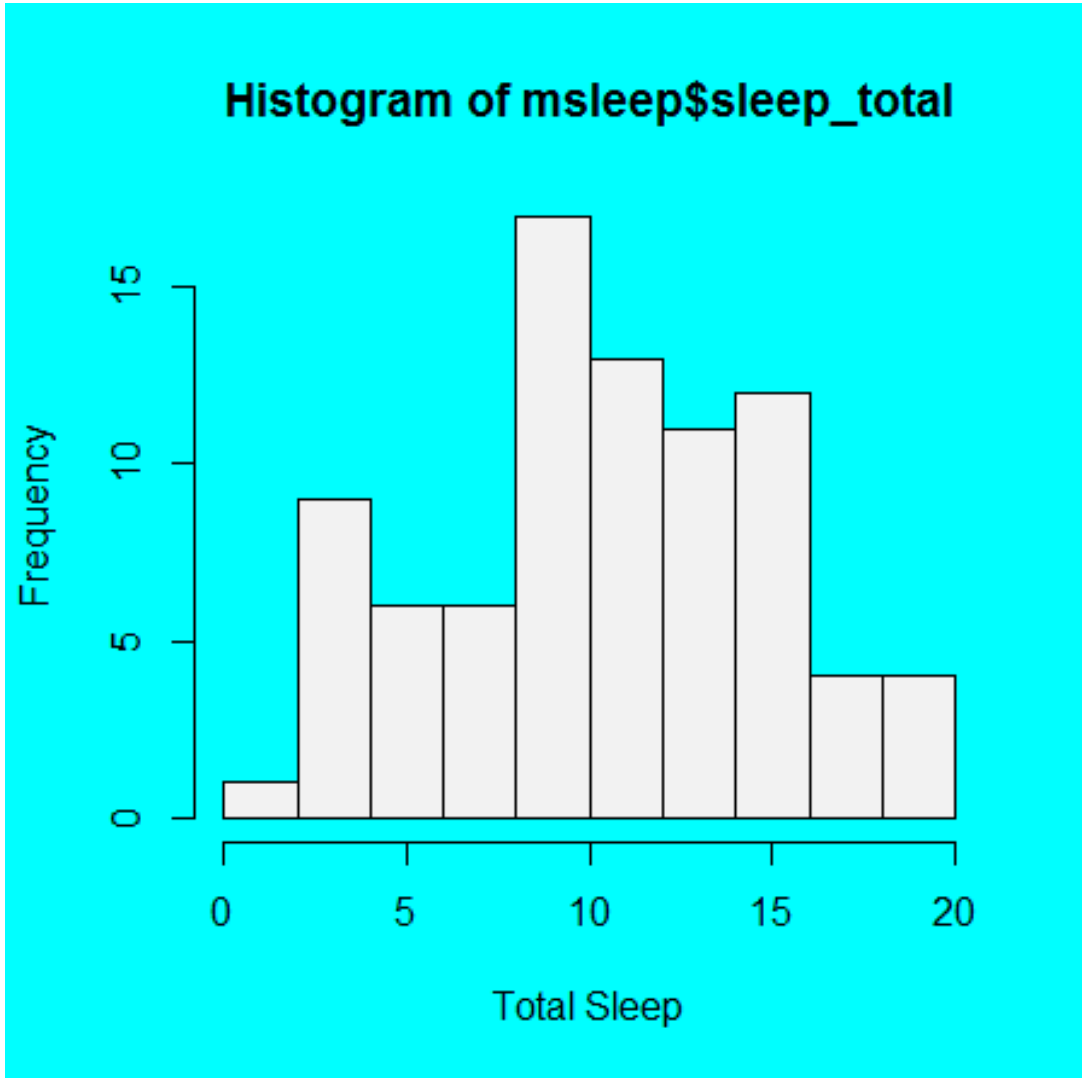
Source: DataMarket - http://bit.ly/2beOvVf

R graphical parameters function

- The **par()** function has information about **graphical parameters** in base R
- Are **primarily** for the **base package**, but **useful** also in **ggplot2**
- **Default settings** are shown when you execute **par()**
- **Some parameters** (e.g. background) can **only** be set using par()
- par() **changes session** parameters, so **take a backup!**
  - **oldPar <- par()**
- You can also use clear all in Rstudio, **but** you will lose all plots!

Changing the background of a plot

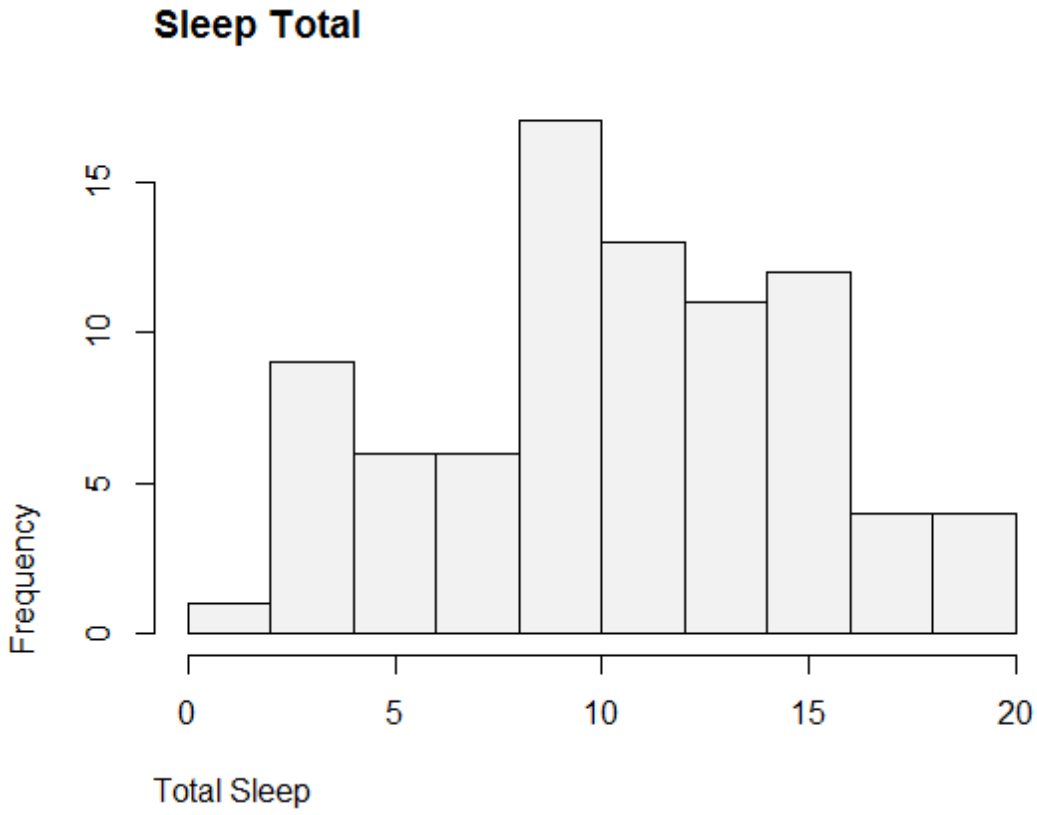
```
oldPar <- par()

par(bg = "cyan")

hist(msleep$sleep_total,
col=gray(0.95), xlab="Total Sleep")

par(oldPar)
```



Change position of text in plot

- Default position: **0.5** = **centre**

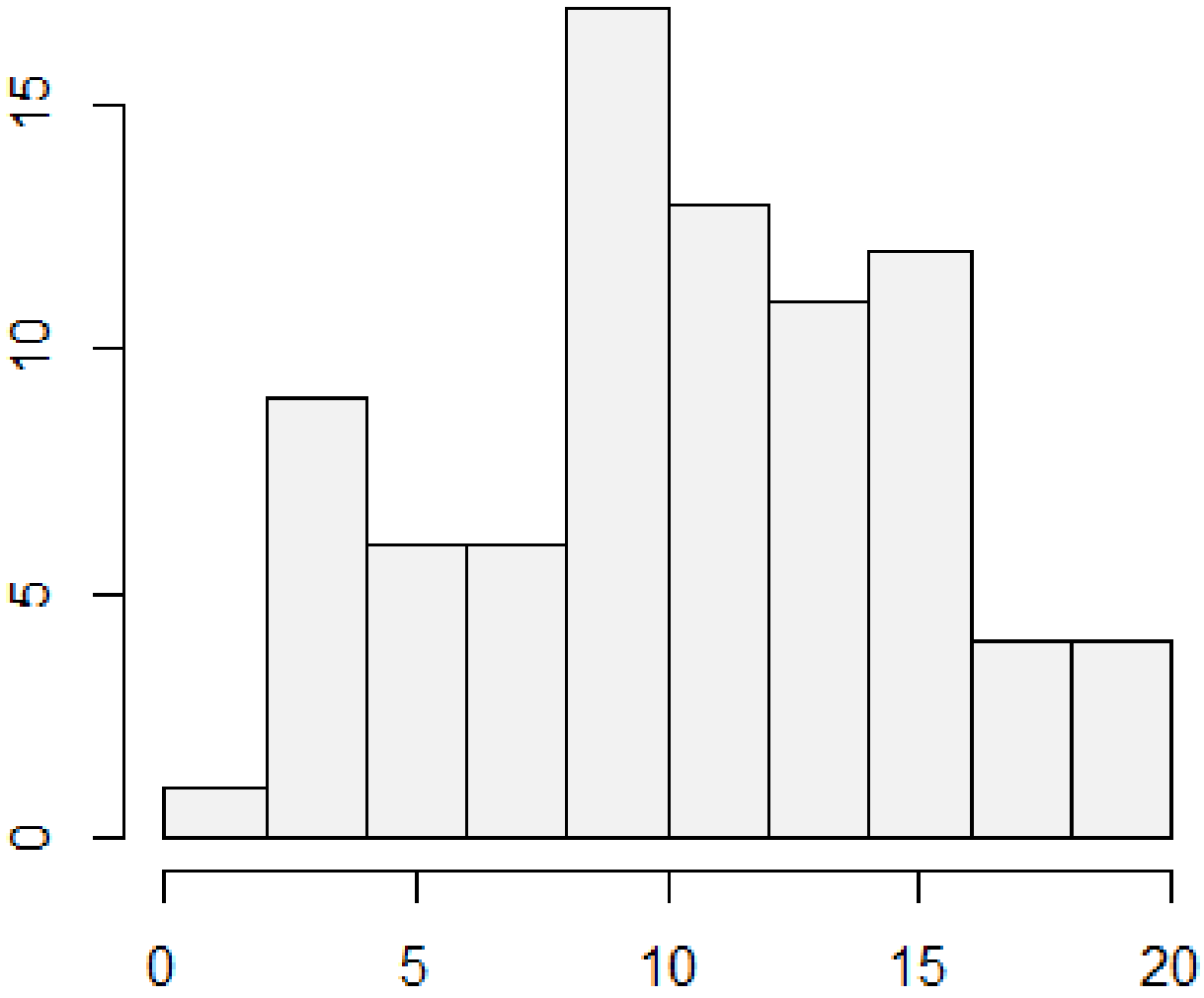
```
hist(msleep$sleep_total, col=gray(0.95),
xlab="Total Sleep", adj=0, main="Sleep Total")

hist(msleep$sleep_total, col=gray(0.95),
xlab="Total Sleep", adj=1, main="Sleep Total")
```



Annotation of scale and main text

- Default: **T** = **show main text and axis scales**

hist(msleep$sleep_total, col=gray(0.95), xlab="Total Sleep", main="Sleep Total", **ann=F**)



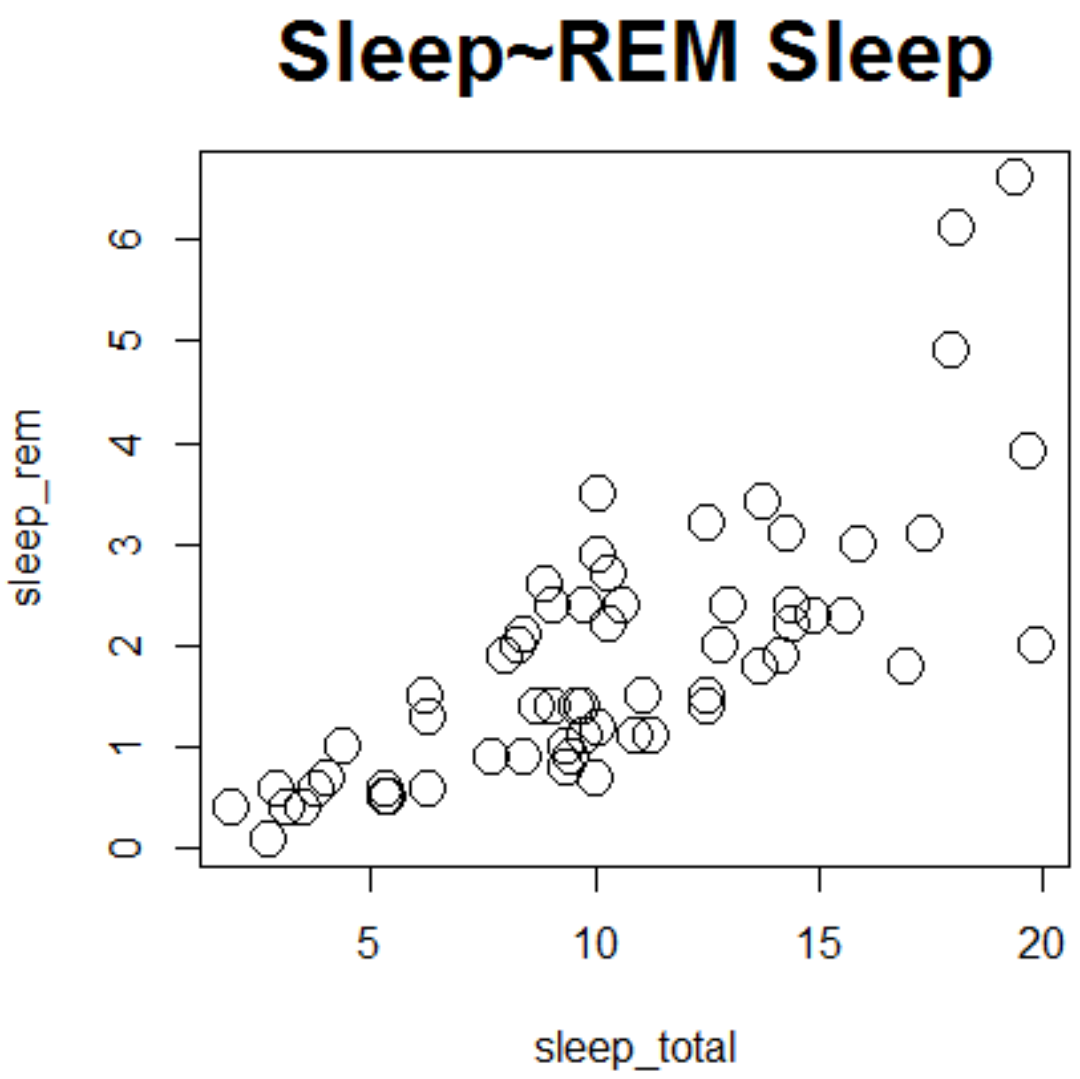
Magnification factor for text

- Default: **1** = **no magnification**

plot(sleep_rem~sleep_total, main="Sleep~REM Sleep", data=msleep, **cex=2**)

plot(sleep_rem~sleep_total, data=msleep, **cex=0.5**)

plot(sleep_rem~sleep_total, data=msleep, **cex.main=2**)



Changing the border type

- Default: **o** = **show box around plot**

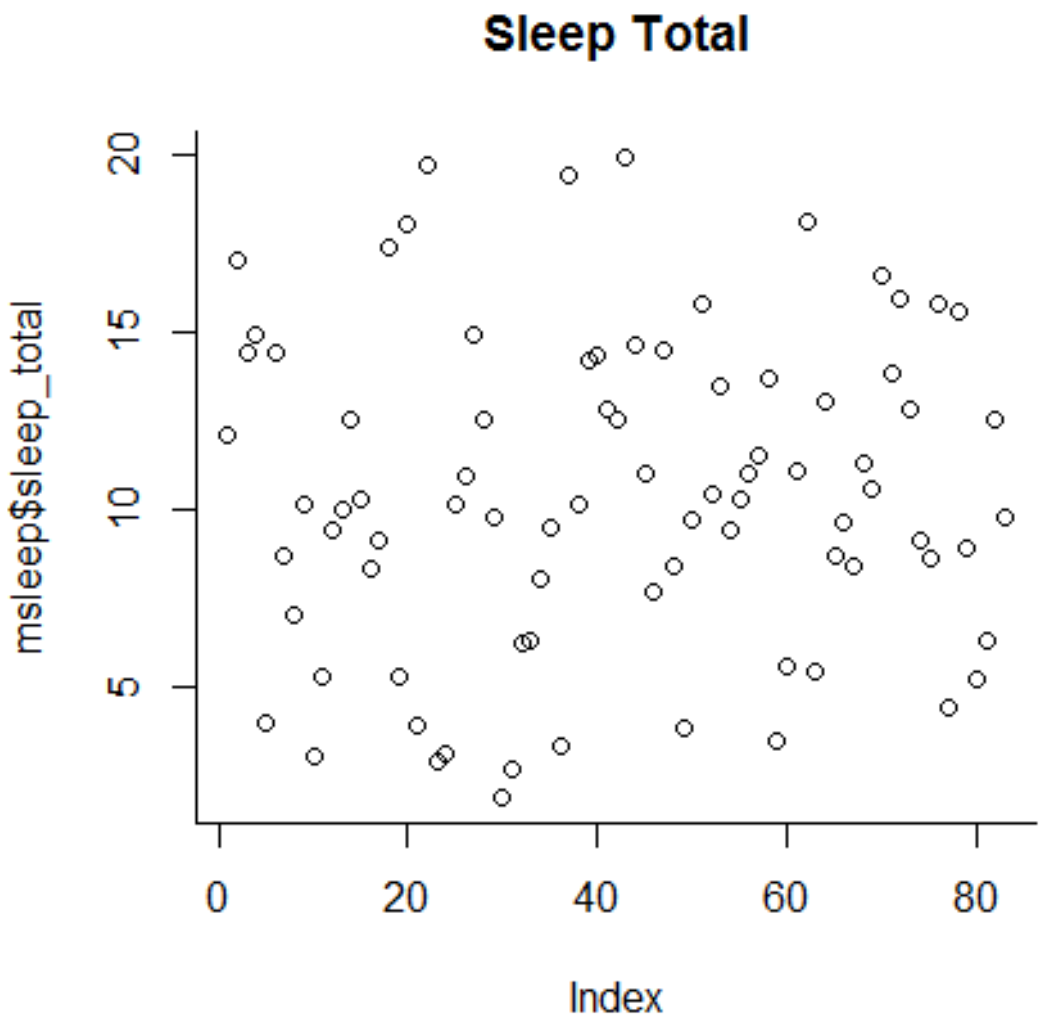
plot(msleep$sleep_total, main="Sleep Total**", bty="n")**

plot(msleep$sleep_total, main="Sleep Total", **bty="c"**)

plot(msleep$sleep_total, main="Sleep Total", **bty="u"**)

plot(msleep$sleep_total, main="Sleep Total", **bty="l"**)

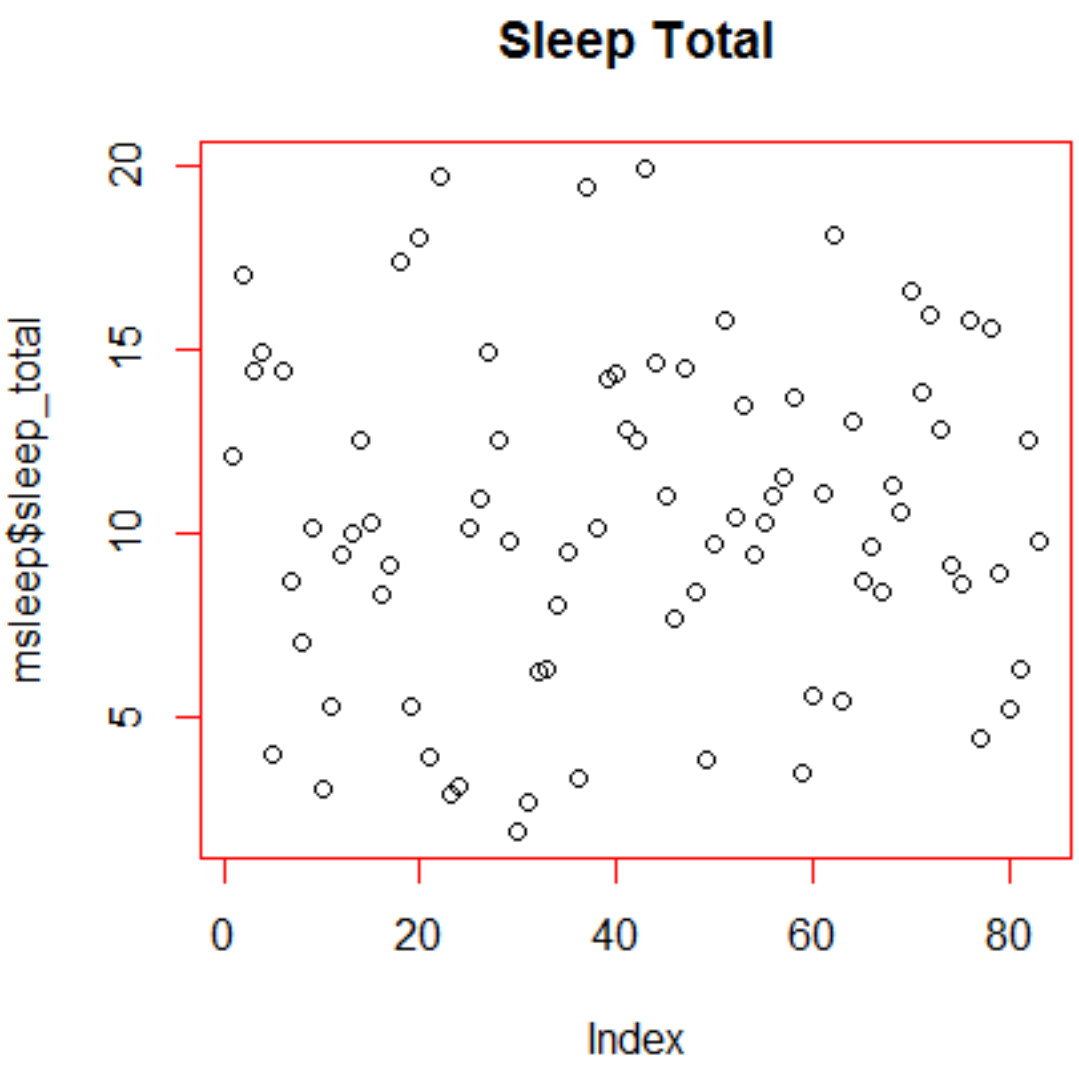
plot(msleep$sleep_total, main="Sleep Total", **bty="7"**)



Changing the foreground

- Default: **white**

plot(msleep$sleep_total, main="Sleep Total", **fg=“red”**)



Getting several plots on one sheet

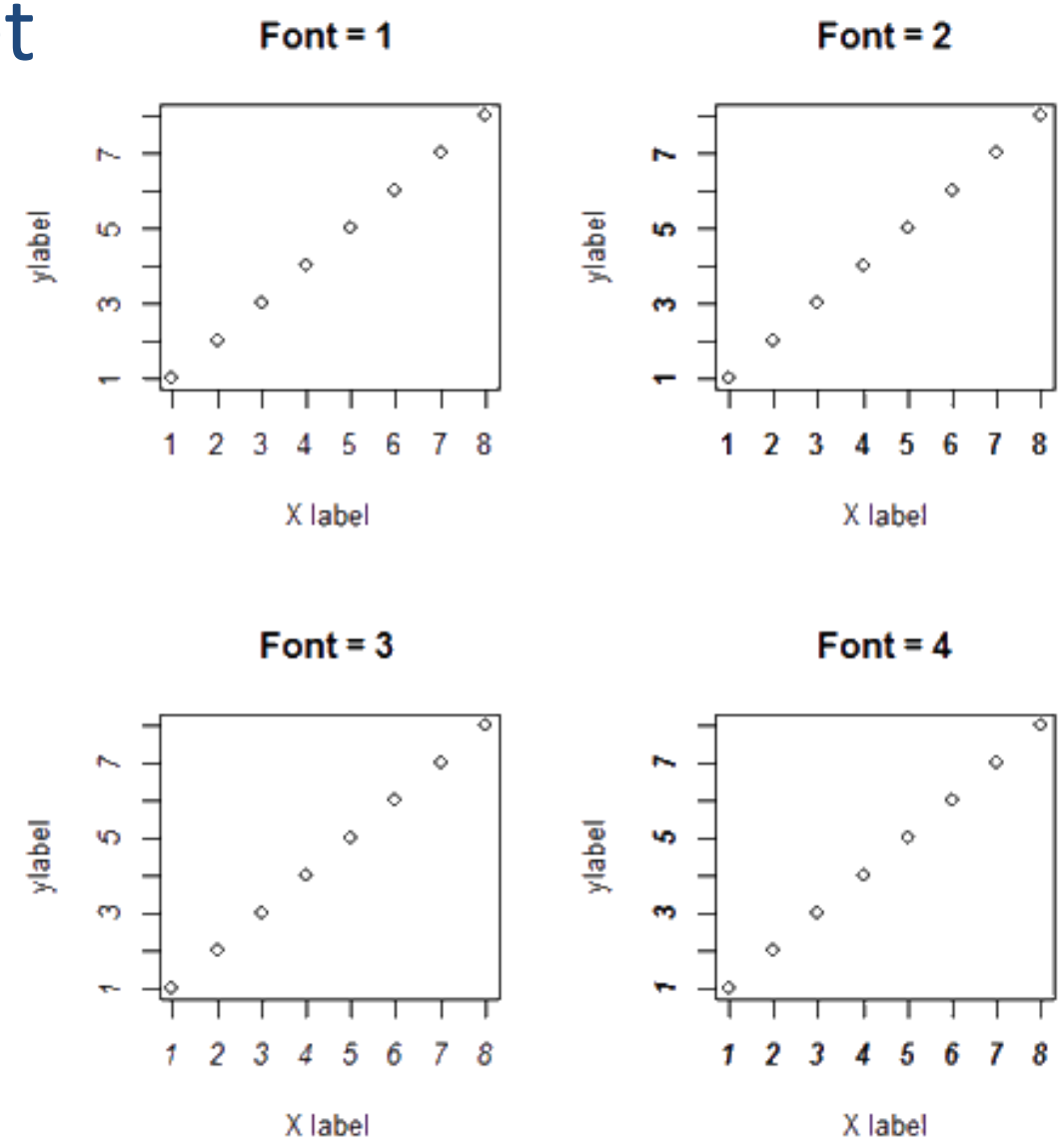
par(mfrow=c(2,2))

plot(1:8, **font=1**,xlab="X label", ylab="ylabel", main="Font = 1")

plot(1:8, **font=2**,xlab="X label", ylab="ylabel", main="Font = 2")

plot(1:8, **font=3**,xlab="X label", ylab="ylabel", main="Font = 3")

plot(1:8, **font=4**,xlab="X label", ylab="ylabel", main="Font = 4")



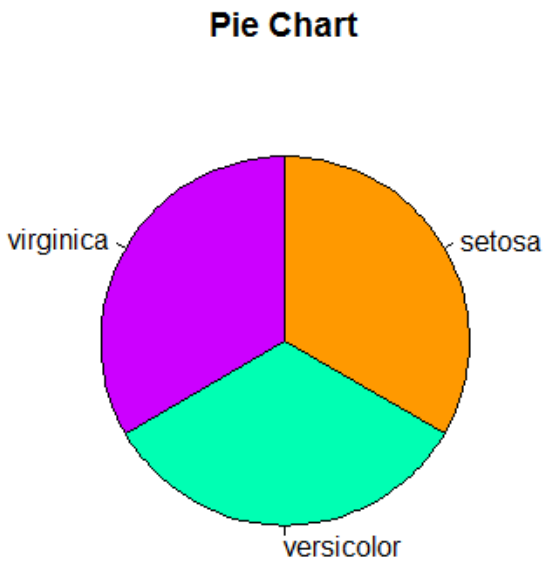
Pie Charts (Adding a legend)

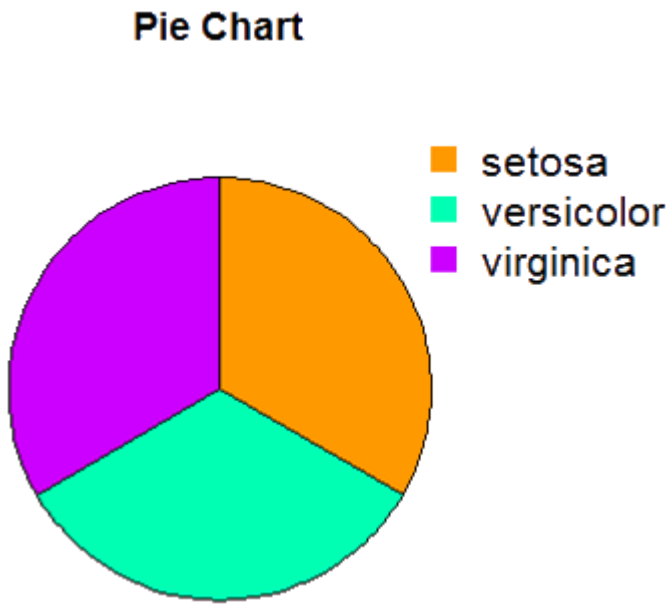
y <- **table(iris$Species)**

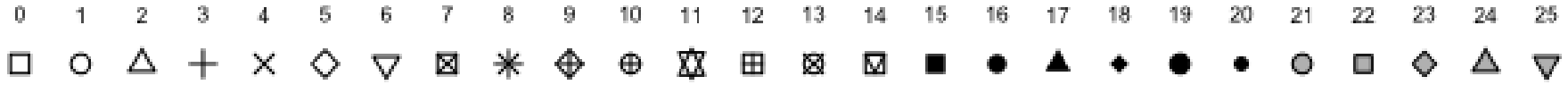
pie(y, **col=rainbow(length(y), start=0.1, end=0.8)**, main="Pie Chart", **clockwise=T**)

pie(y, col=rainbow(length(y), start=0.1, end=0.8), **labels=NA**, main="Pie Chart", clockwise=T)

legend("topright", legend=row.names(y), bty="n", **pch=15**, col=rainbow(length(y), start=0.1, end=0.8), ncol=1)



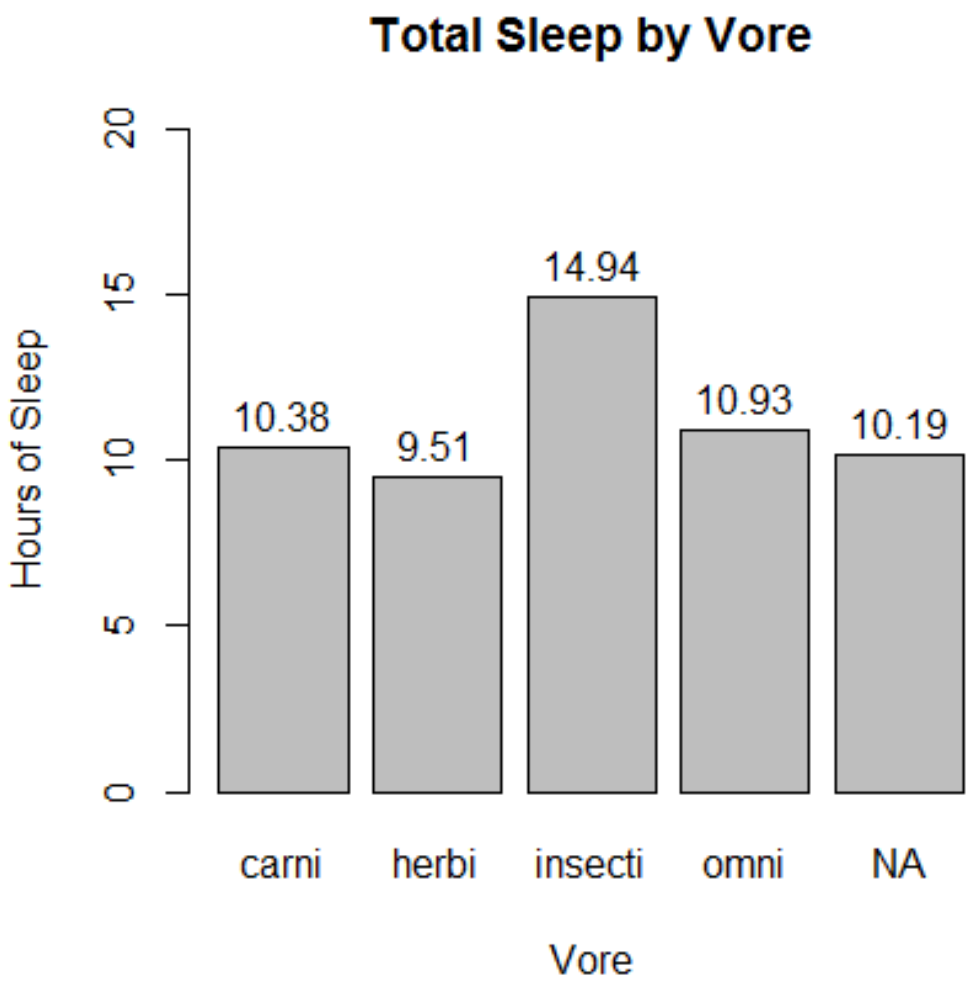




Barplots

```
midpoints <- barplot(sleep.groupedby.vore$avg_sleep,
main="Total Sleep by Vore", xlab="Vore“,
ylab="Hours of Sleep“, names.arg =
c(as.vector(sleep.groupedby.vore$vore[1:4]), "NA"))

text(midpoints,
y=sleep.groupedby.vore$avg_sleep+1.0,
labels=round(sleep.groupedby.vore$avg_sleep,2))
```

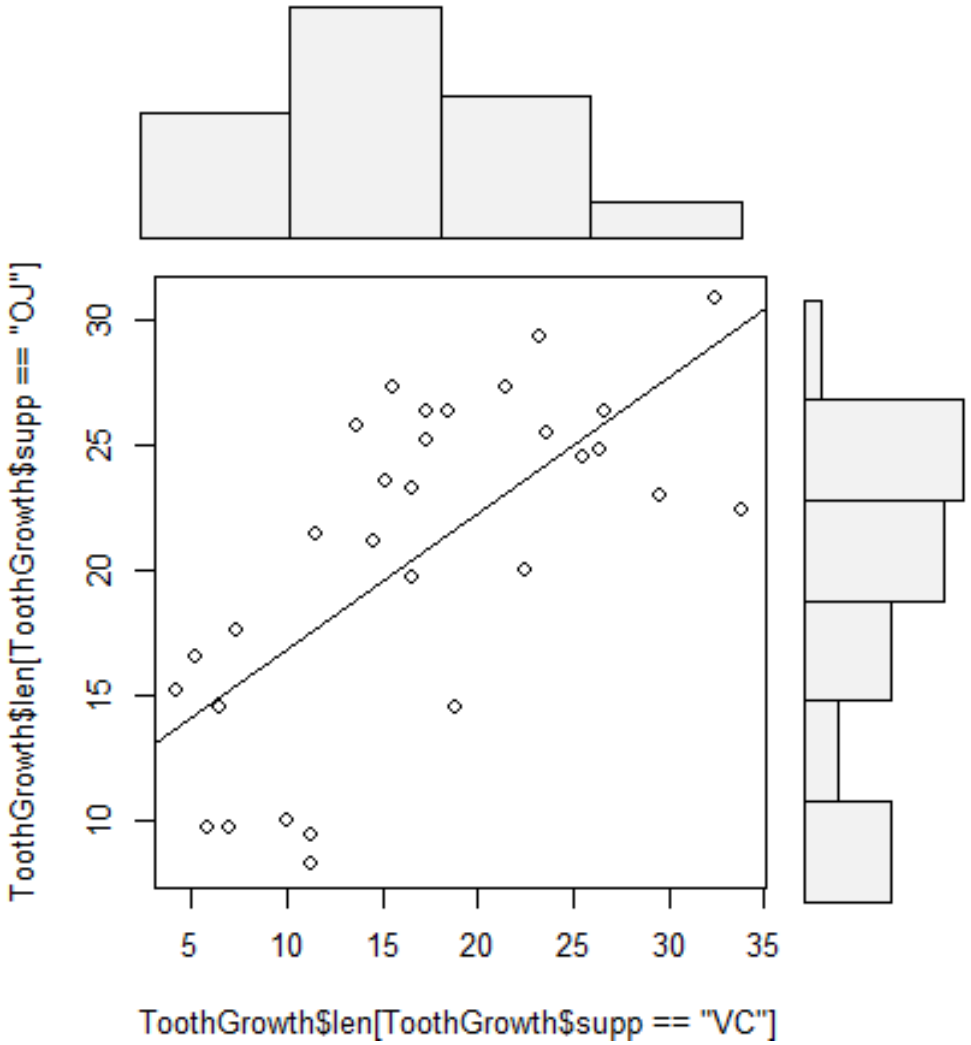
```
(sleep.groupedby.vore <- msleep %>% group_by(vore)
%>% summarise(avg_sleep = mean(sleep_total),
total=n()))
```



Bivariate Histogram

install.packages("UsingR")
library(UsingR)

scatter.with.hist(ToothGrowth$len[ToothGrowth$supp == 'VC'], ToothGrowth$len[ToothGrowth$supp =='OJ'])



Saving Graphics to Files

- **jpeg("test.jpeg"); plot(1:10, 1:10); dev.off()**
  - After the 'jpeg("test.jpeg")' command **all graphs are redirected** to the file "test.jpeg" in JPEG format.
  - To export images with the **highest quality**, the default setting "**quality = 75**" needs to be changed to 100.
  - The **actual image** data are **not written** to the file until the 'dev.off()' command is executed!

- **pdf("test.pdf"); plot(1:10, 1:10); dev.off()**
  - Same as above, but for pdf format. The pdf format provides often the best image quality, since it scales to any size without pixelation.

Saving Graphics to Files

- **`png("test.png"); plot(1:10, 1:10); dev.off()`**
  - Same as the previous examples, but for png format.

- **`postscript("test.ps"); plot(1:10, 1:10); dev.off()`**
  - Same as the previous examples, but for PostScript format.

Lesson 6: Wrap-up

- Used R Base package for visualisations
  - Scatter plots
  - Box plots
  - Time series
  - Histograms
  - Pie Charts
  - Barcharts

  - Simple tweaking

  - Saved Visualisations to file

Lesson 7

Cool plots with ggplot2

Lesson 7: Objectives

- Use R ggplot2 package for visualisations
  - Introduction to ggplot2
  - Use qplot for quick plots
  - Bar Graphs
  - Scatter plots
  - Adding aesthetics
  - Flipping a plot
  - Boxplots

ggplot2

- Brief History and Background
  - An R **package** developed by Hadley Wickham in 2010
  - Based on **Grammar of Graphics** (Wilkinson, 2005)
  - A library with a set of **independent components** that can be **composed** in various ways
  - Plots built **iteratively** and edited later
  - Has a **carefully chosen** set of defaults, and a powerful theming system

ggplot2 function

- ggplot2 has **two main** functions:

- qplot
  - Short for **quickplot**, a **simple function** to plot data
  - qplot uses only **one dataset** and only **one aesthetic**

- ggplot
  - A **more expressive** library that is preferred for complex plots

Installing and using ggplot2

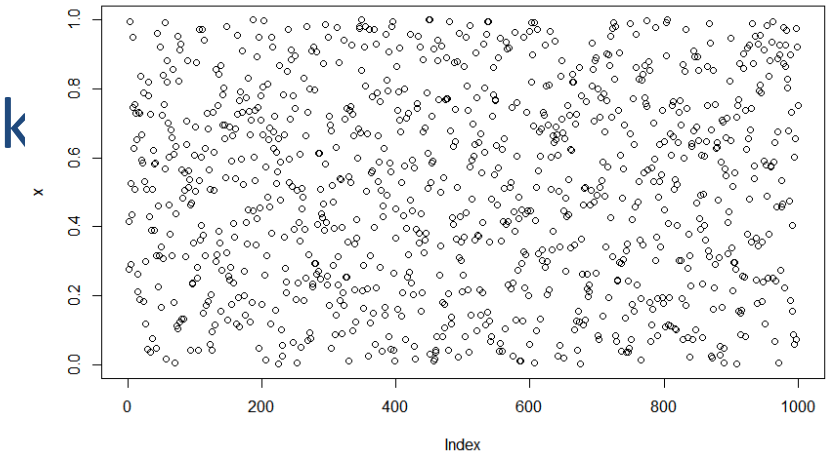
```
### install & load ggplot library
install.packages("ggplot2")
library("ggplot2")
```

Simple scatter plot

x <- runif(1000)

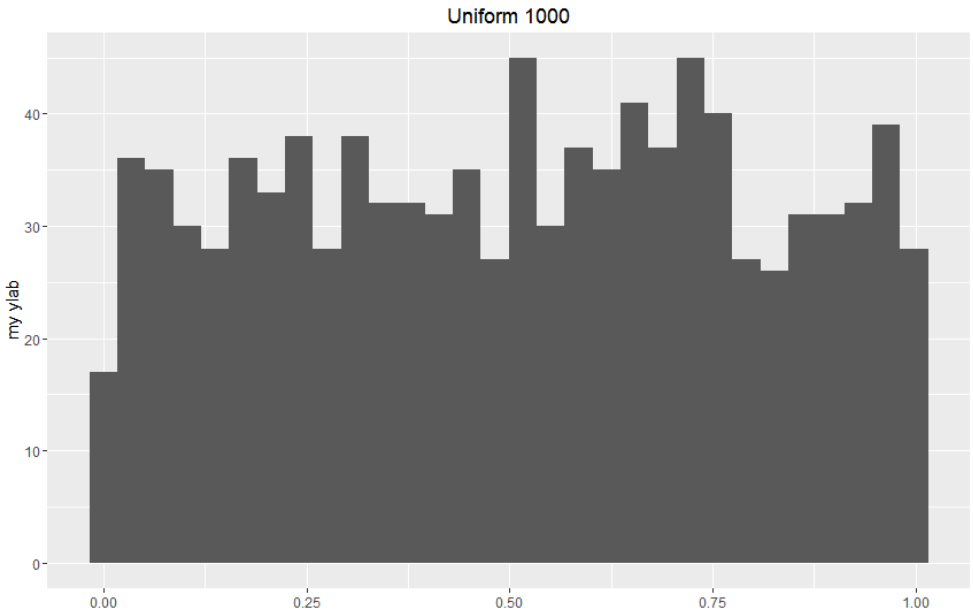
Using R base pack

plot(x)



#Using ggplot2

qplot(x)



#With some familiar syntax

qplot(x, main="Uniform 1000", ylab="my ylab", xlab="my xlab")

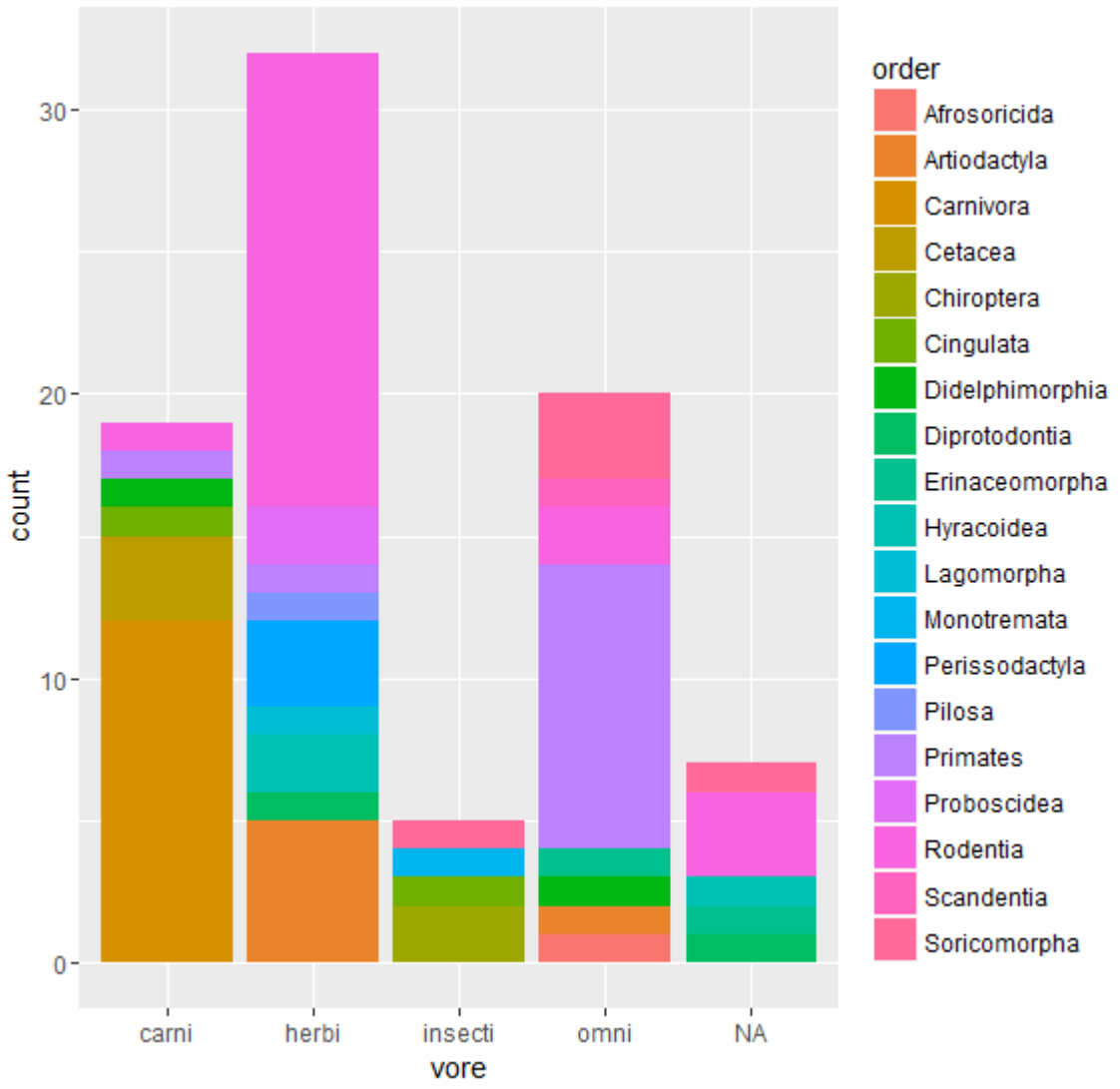
Quickplot and ggplot!

qplot histogram

qplot(vore, **fill**=order, **data**=msleep, **geom="bar"**)

ggplot histogram -> same output

ggplot(msleep, **aes**(vore, **fill**=order)) + **geom_bar()**

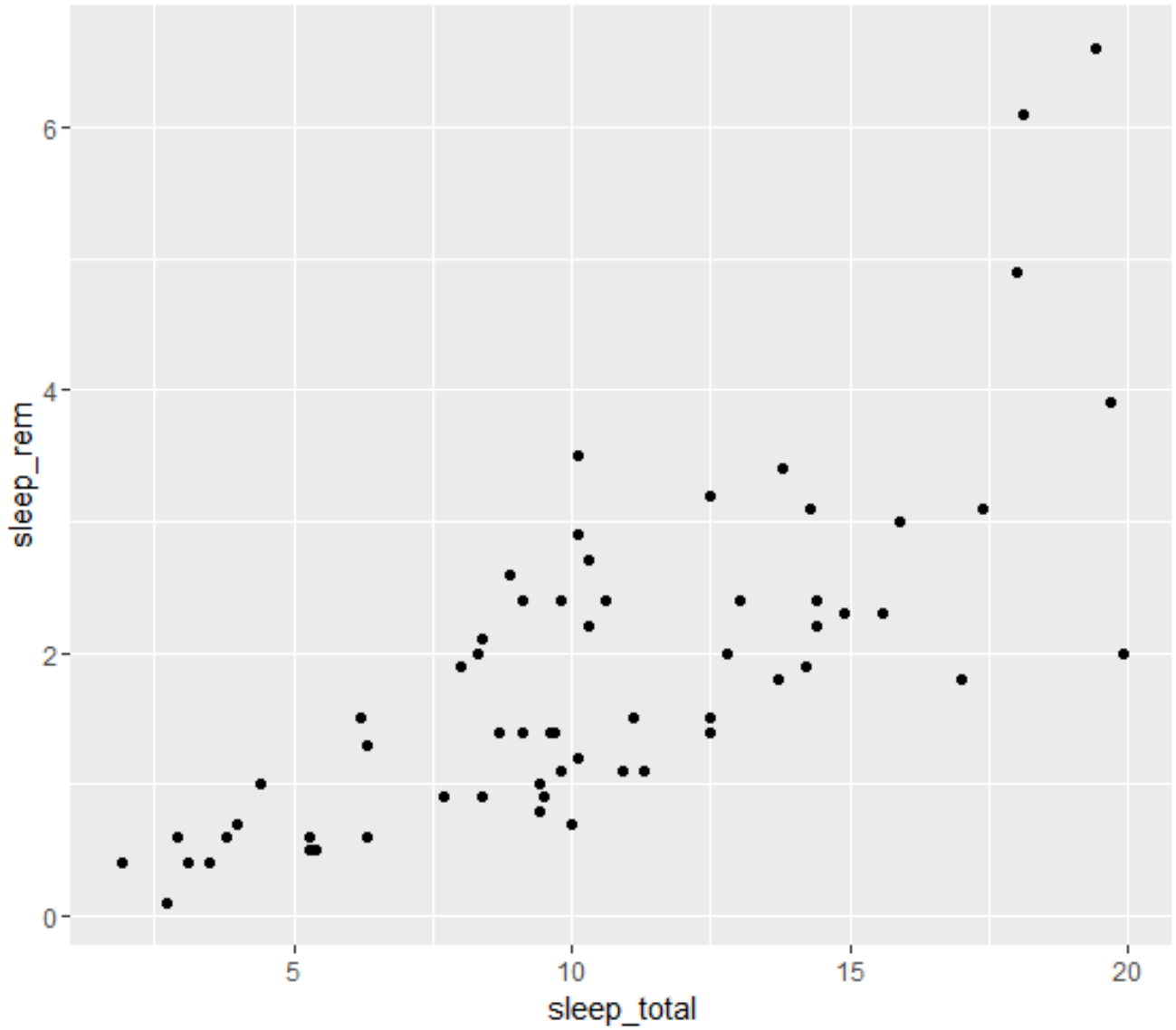


Using qplot

how to use qplot

scatterplot

qplot(sleep_total, sleep_rem, data=msleep)

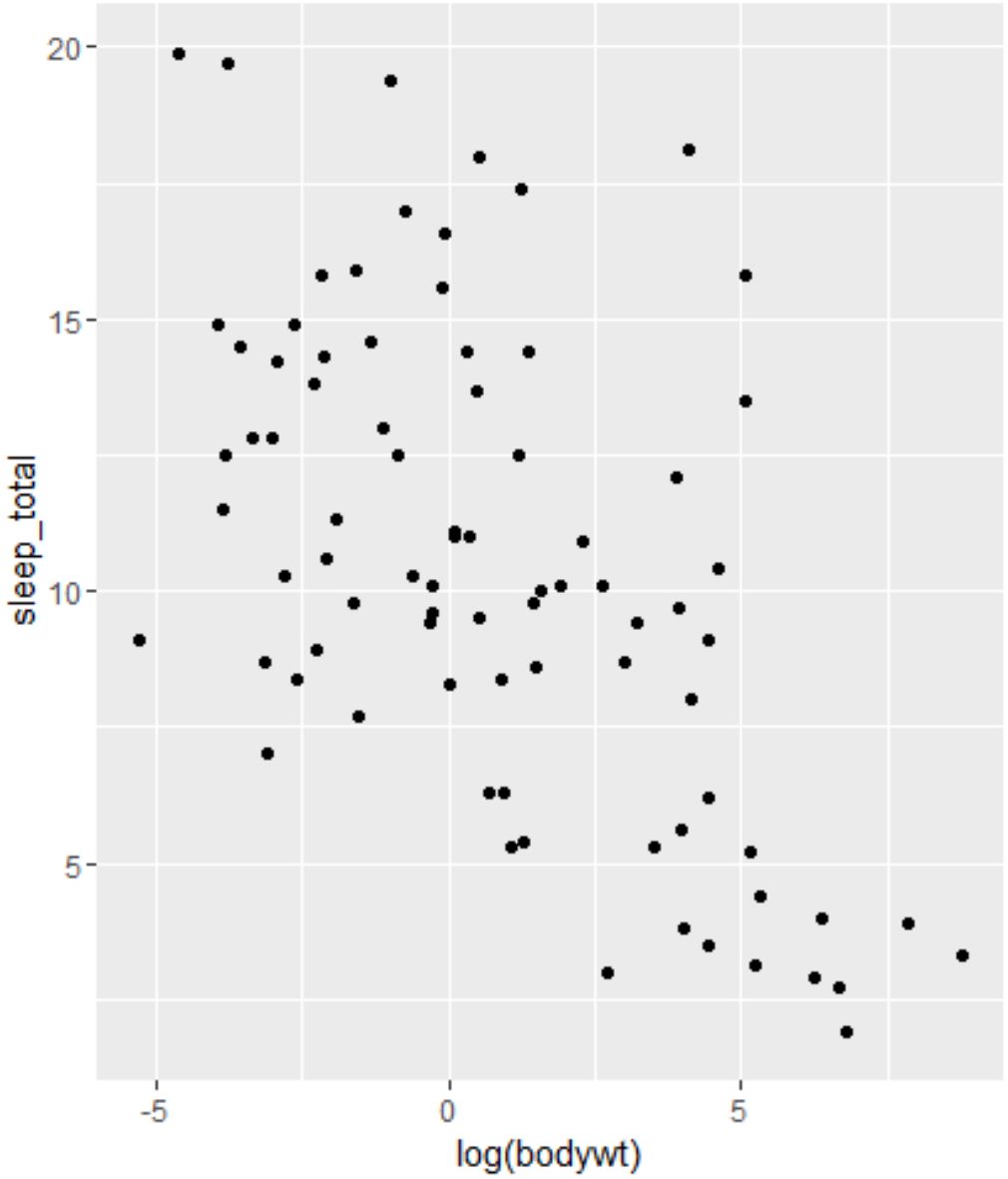


Using qplot

Applying function transformations

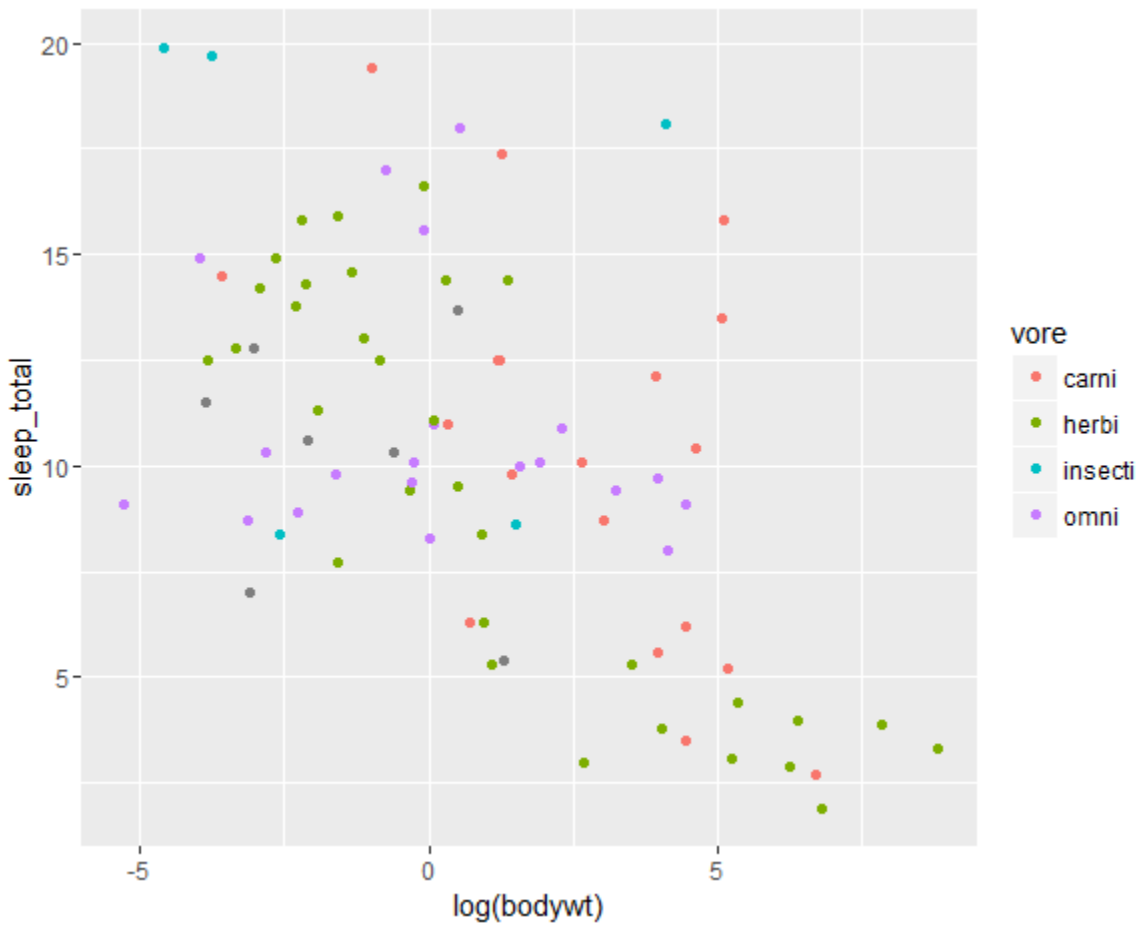
qplot(**log(bodywt)**, sleep_total, data=msleep)

log() computes the (natural) logarithm of a number ($\log_{10}$)



Adding some aesthetics

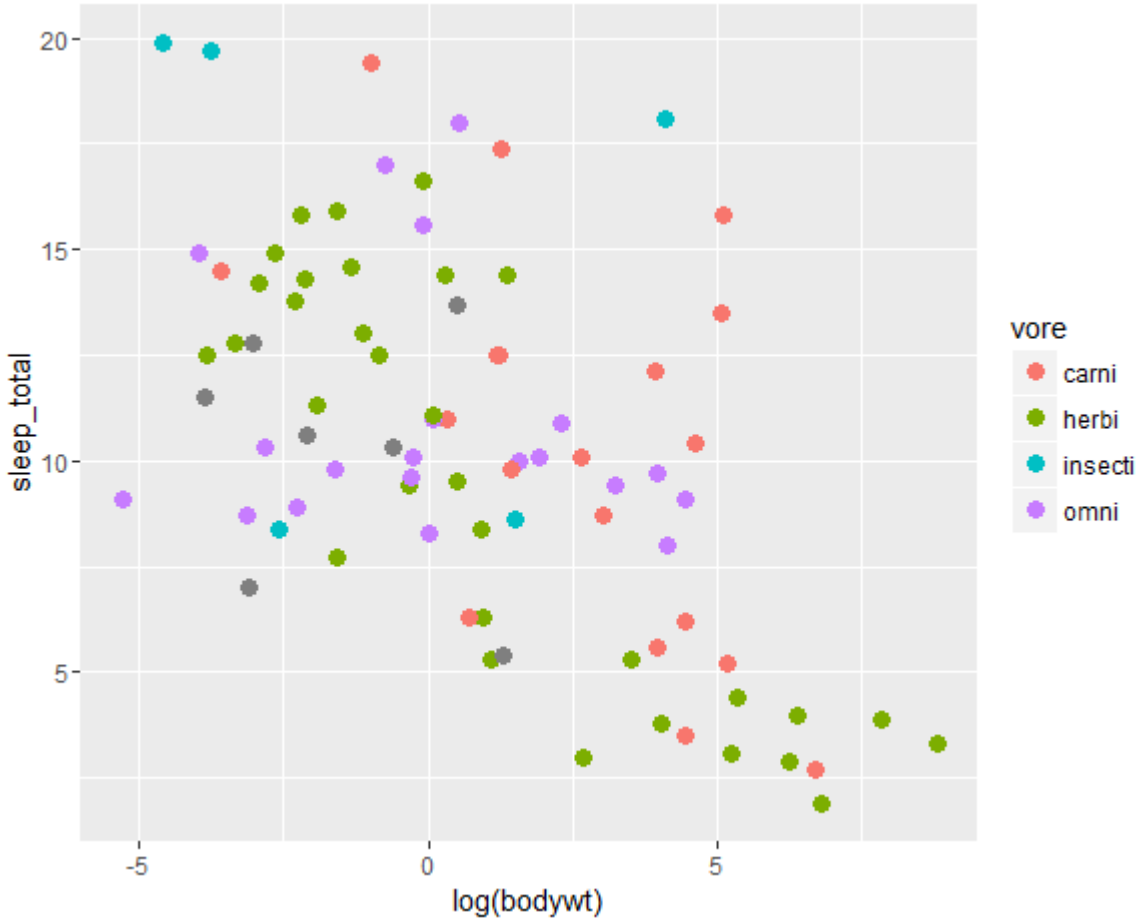
qplot(log(bodywt), sleep_total, data=msleep, **color=vore**)



Change the size of points

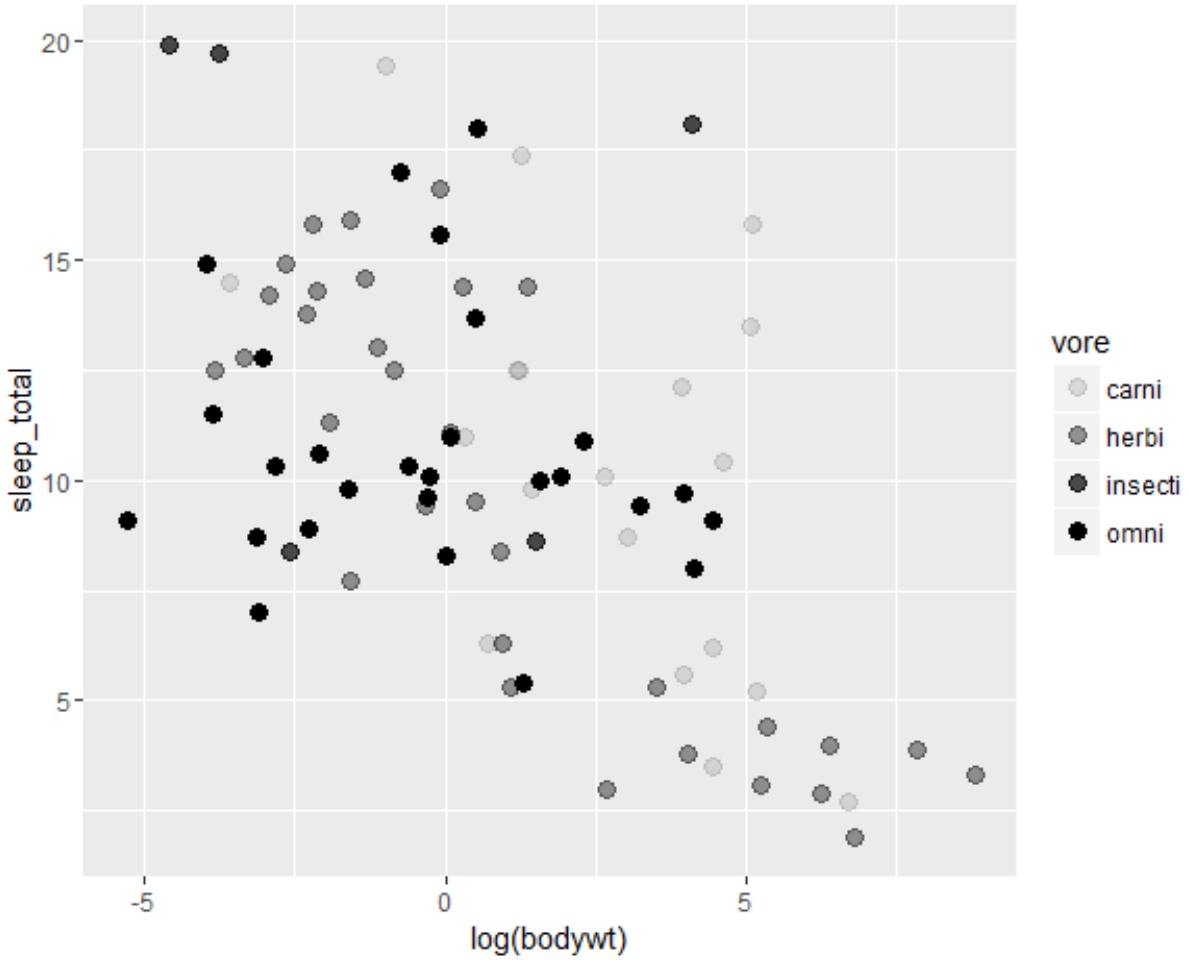
qplot(log(bodywt), sleep_total, data=msleep, color=vore, **size=I(3)**)

- **I()** is the "as is" function, it tells R to treat the value as is
- What happens if you omit I?



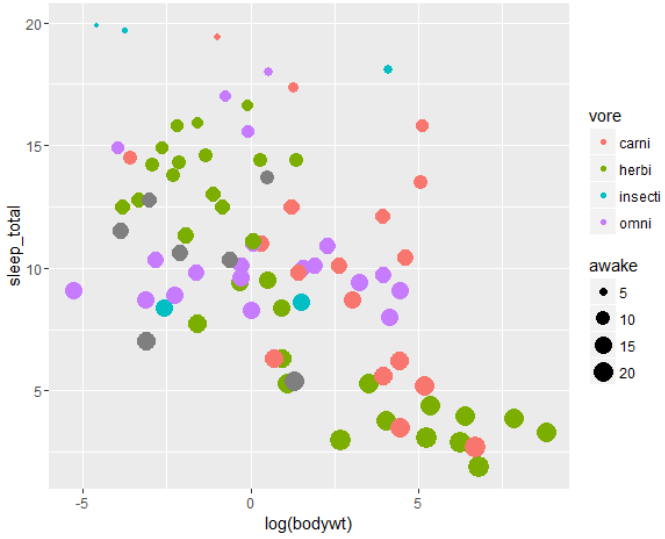
Using alpha blending

qplot(log(bodywt), sleep_total, data=msleep, **alpha=vore**, size=I(3))

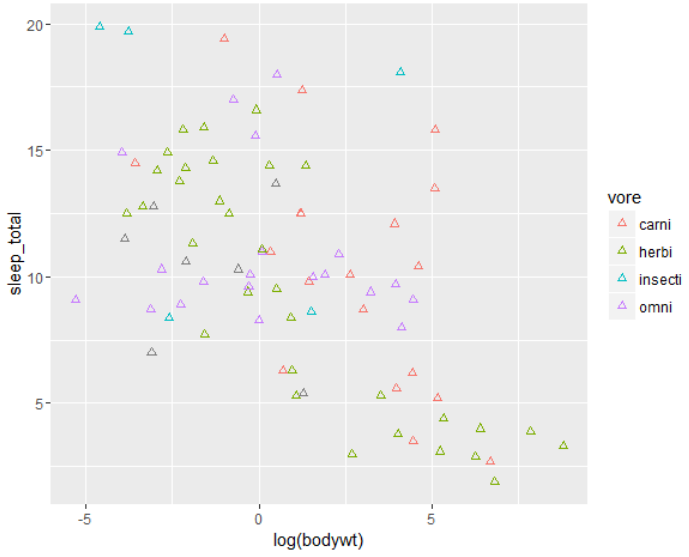


Combining Mappings

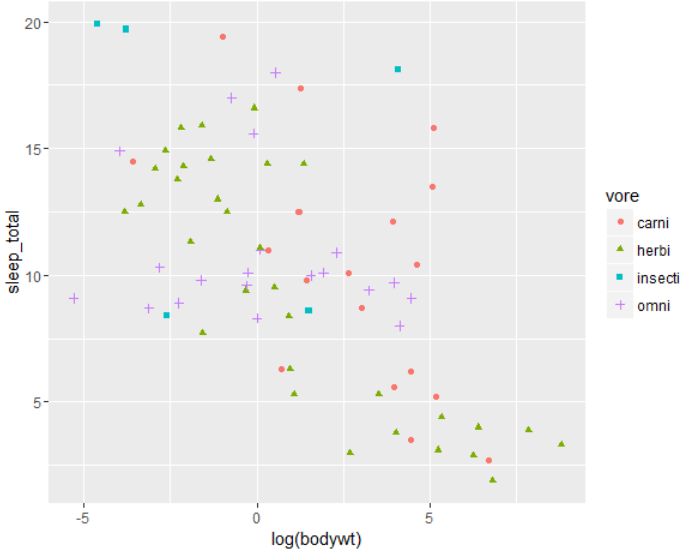
qplot(log(bodywt), sleep_total, data=msleep, color=vore, size=awake)



qplot(log(bodywt), sleep_total, data=msleep, color=vore, shape=I(2))

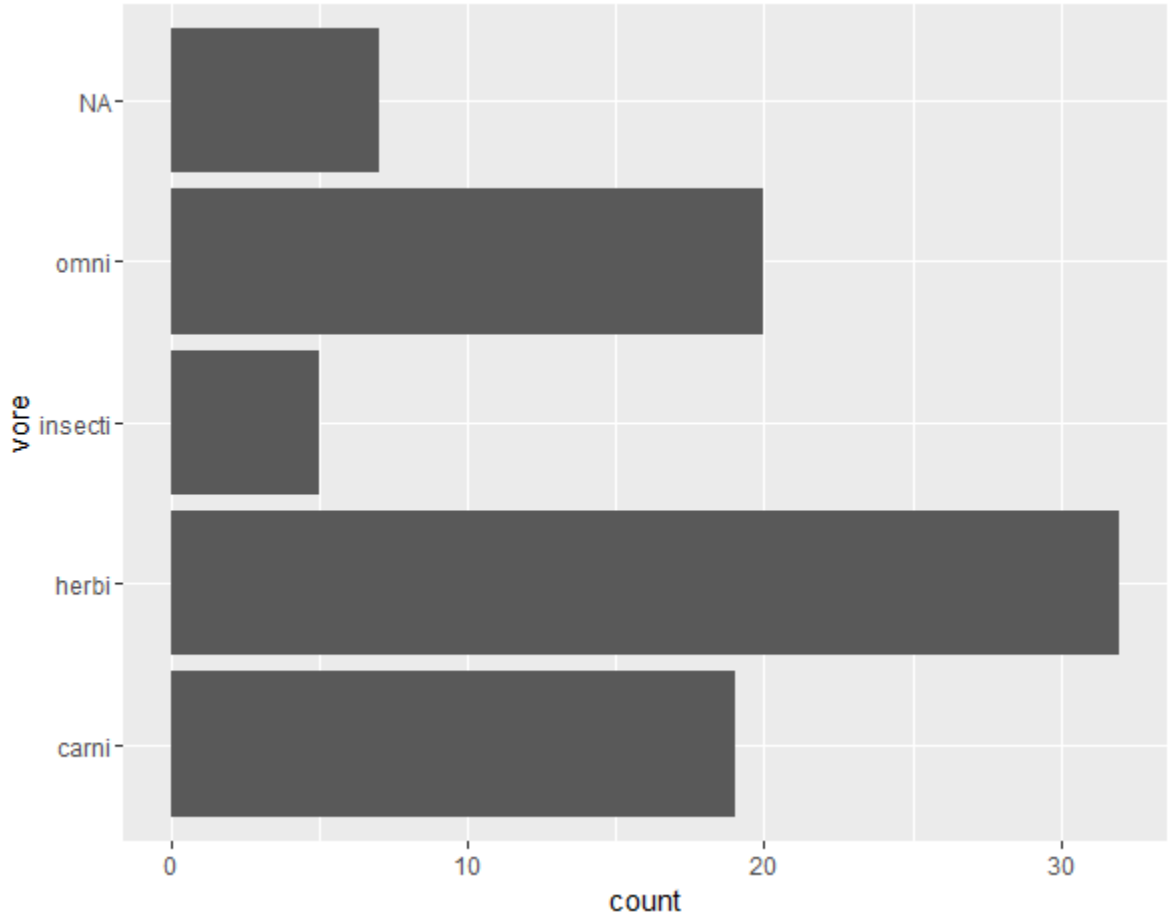


qplot(log(bodywt), sleep_total, data=msleep, color=vore, shape=vore, geom="point")



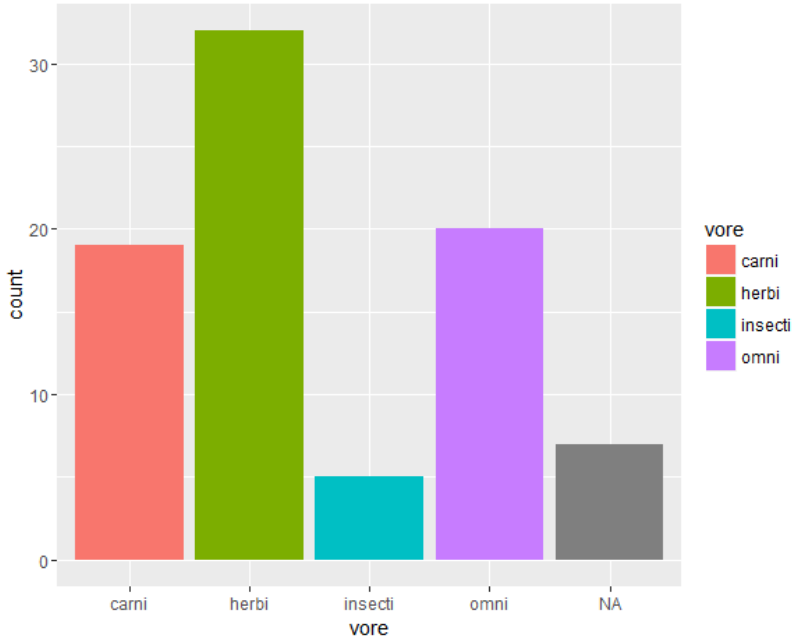
Flipping a plot

```
qplot(vore, data=msleep,
geom="bar") + coord_flip()
```

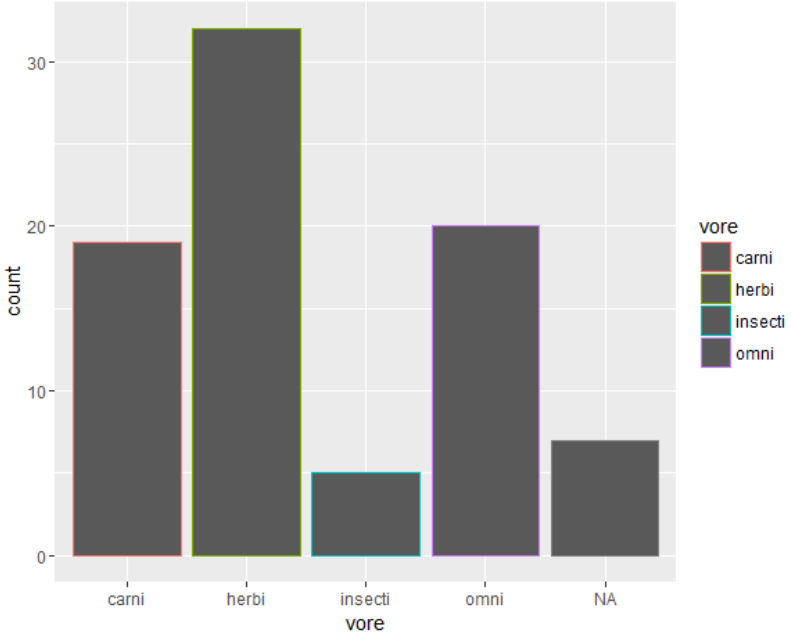


Colouring and Filling bars

qplot(vore, data=msleep, geom="bar", fill=vore)

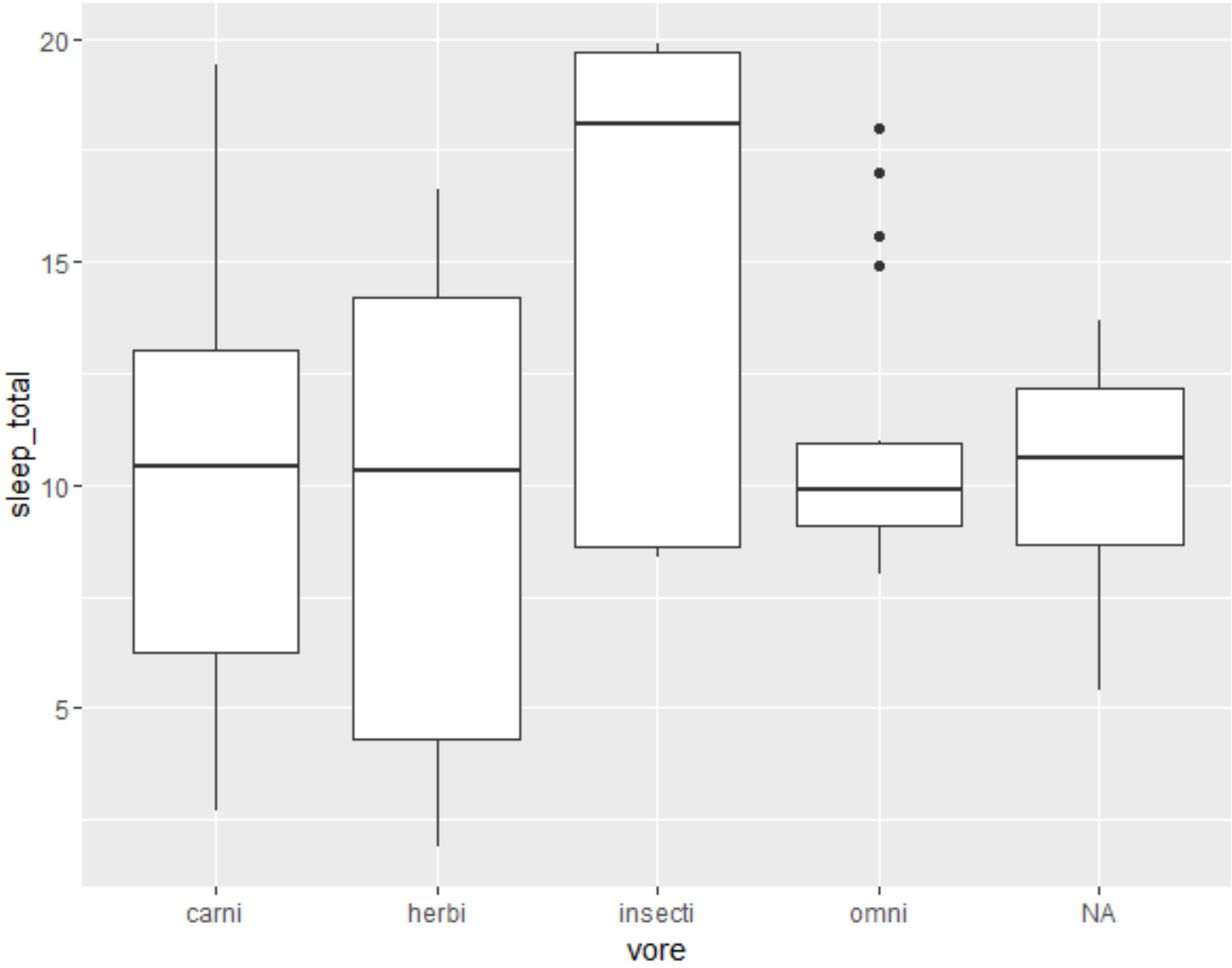


qplot(vore, data=msleep, geom="bar", color=vore)



Boxplots

qplot(vore, sleep_total, data=msleep, geom="boxplot")



Lesson 7: Wrap-up

- Used R ggplot2 package for basic visualisations
  - Introduced ggplot2
  - Used qplot for quick plots
  - Bar Graphs
  - Scatter plots
  - Added aesthetics
  - Flipped a plot
  - Boxplots

Lesson 8

Advanced plots with ggplot2

Lesson 8: Objectives

- Use R ggplot2 package for advanced visualisations
  - Use the **with** function
  - Exploit implicit order in the data
  - Plot categorical and quantitative variables
  - Summarise of ggplot geometric objects

Download using browser

- Download students.csv (using a browser) from http://goo.gl/anF2Lo

- Load the following datasets:

students <- read.csv(**file.choose()**) # students.csv

Analyse the data

```
str(students)

dim(students)

nrow(students)

ncol(students)

summary(students)
```

With – Refer so same variable in same command

- Get the total number of siblings for each student

```
siblings <- with(students, Brothers+Sisters)
print(siblings)
```

Same as

```
Siblings2 <- students$Brothers + students$Sisters
```

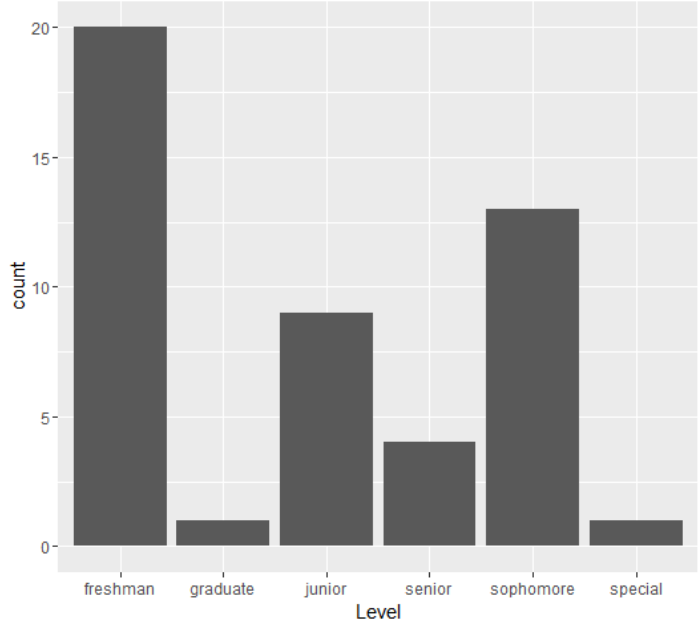
Plot a categorical variable against count

Syntax

- **ggplot(data, aesthetics) + geometric representation layer**

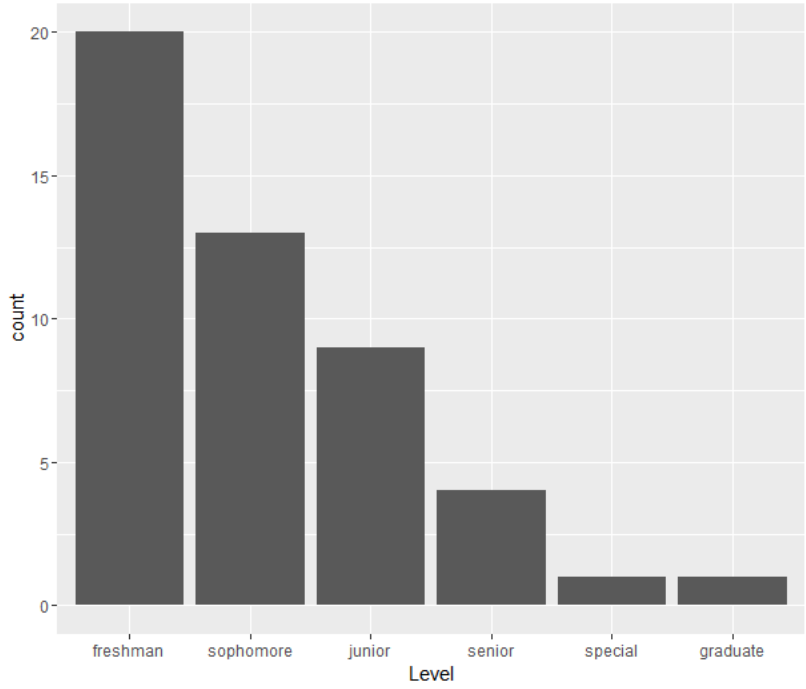
- aesthetics = characteristics the plot should have – x-coords, y-coords, colour, shape, etc.

ggplot(students, aes(x = Level)) + geom_bar()



Improving the previous graph

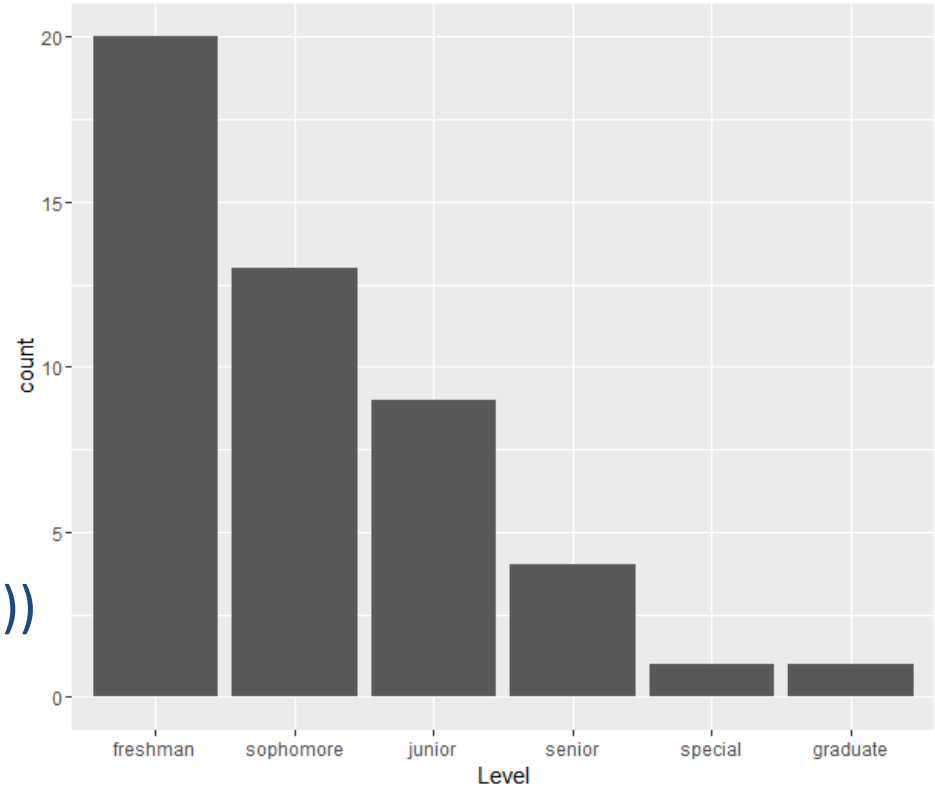
- Ideally, try to exploit the natural properties of the variables you want to plot
- Level has a natural partial order from freshman to senior, followed by special and graduate
- To achieve this, use the reorder function to reassign Levels in the graph



Let's see how …

Improving the previous graph

```
orderedLevels <- rep(0, nrow(students))

orderedLevels[with(students, Level == "freshman")] = 1
orderedLevels[with(students, Level == "sophomore")] = 2
orderedLevels[with(students, Level == "junior")] = 3
orderedLevels[with(students, Level == "senior")] = 4
orderedLevels[with(students, Level == "special")] = 5
orderedLevels[with(students, Level == "graduate")] = 6

students$Level <- with(students, reorder(Level, orderedLevels))
rm(orderedLevels)
ggplot(students, aes(x = Level)) + geom_bar()
```

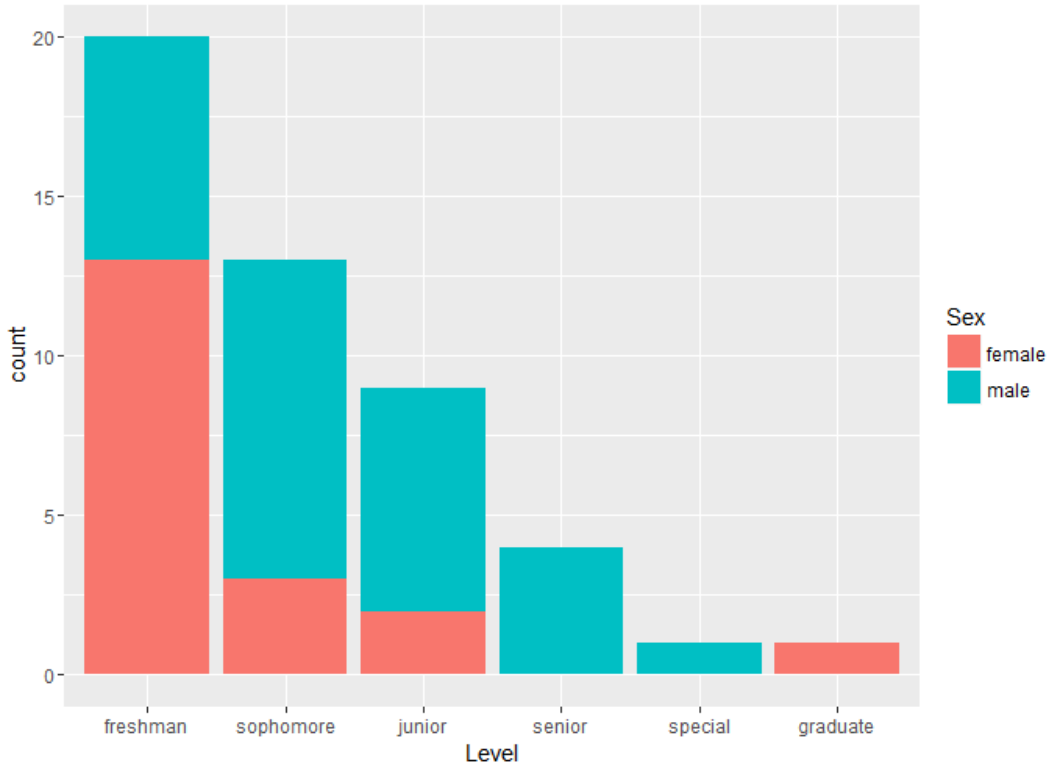


Bar plots with two categorical variables

- You want to examine the gender distribution by level

with(students, table(Sex, Level))

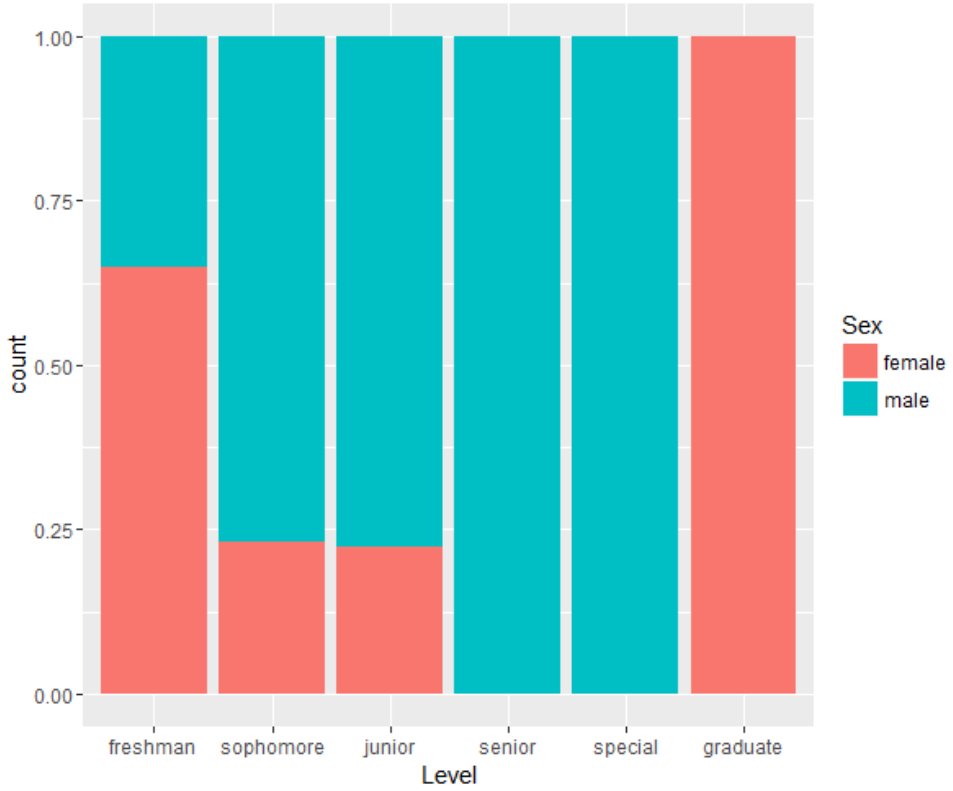
ggplot(students, aes(x=Level, fill=Sex)) +
geom_bar()



Bar plots with two categorical variables

- The previous graph is a little difficult to interpret as it deals exclusively with counts

ggplot(students, aes(x=Level, fill=Sex)) +
geom_bar(position = "fill")

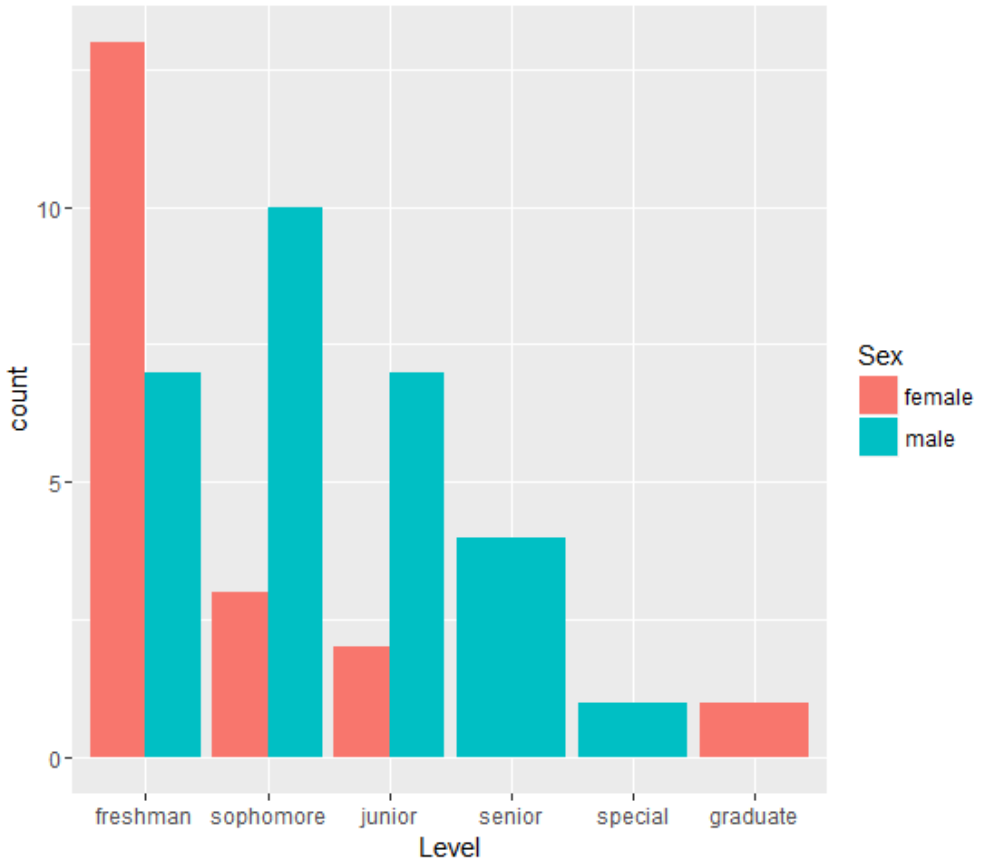


Bar plots with two categorical variables

- Improving further, we can show each gender separately to make the visualisation clearer

- Note what happens where there is only one gender in a particular level

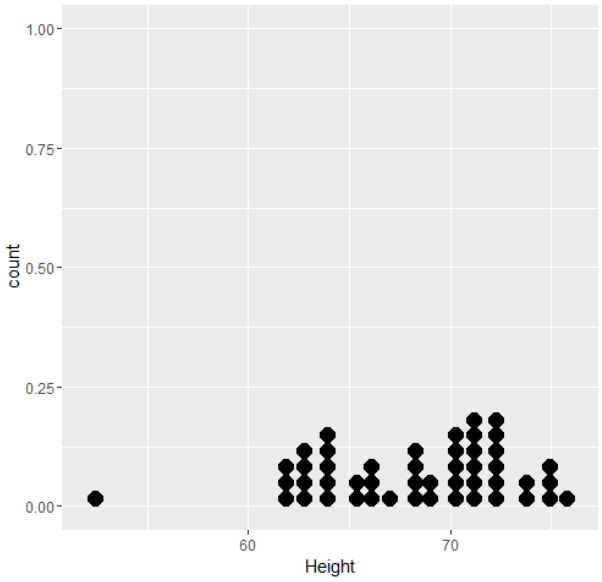
```
ggplot(students, aes(x=Level, fill=Sex)) +
geom_bar(position = “dodge")
```



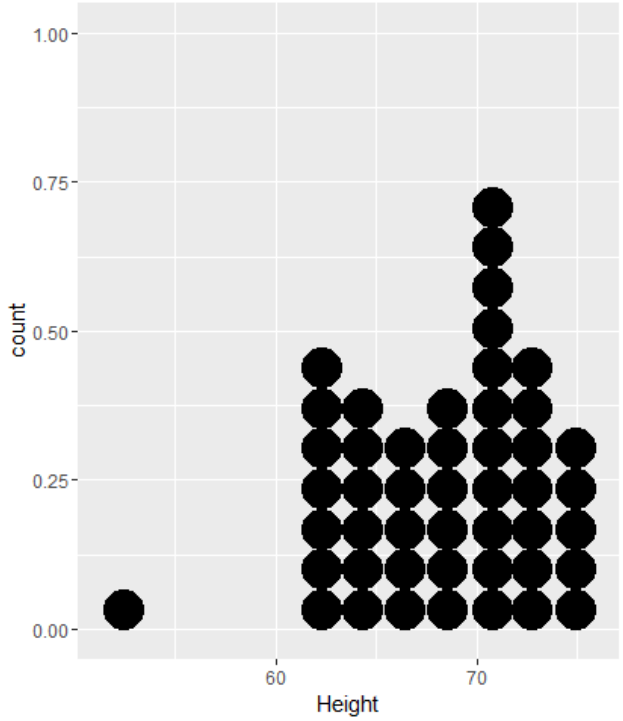
Plotting Quantitative Variables

- Suitable for small datasets

```
ggplot(students, aes(x = Height)) +
geom_dotplot()
```



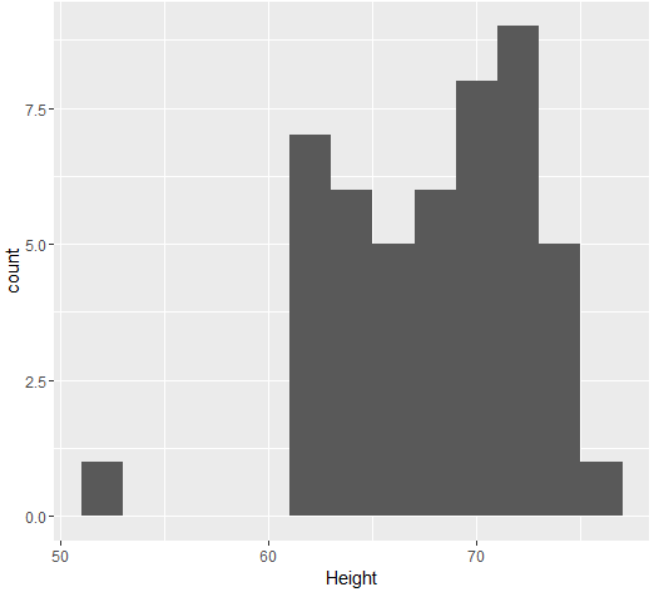

```
ggplot(students, aes(x = Height)) +
geom_dotplot(binwidth = 2)
```



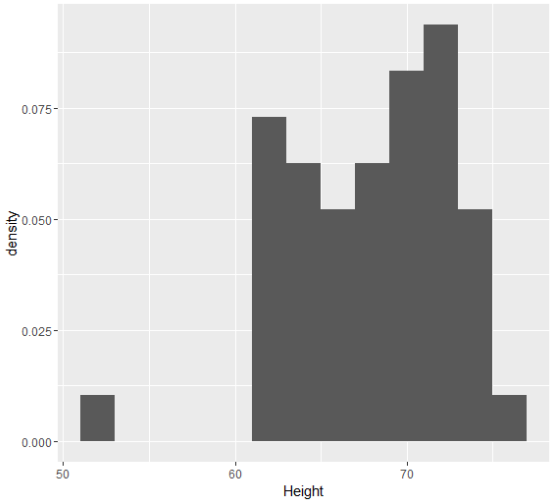
Plotting Quantitative Variables

- Use histograms for larger datasets
- Set binwidth manually is useful

```
ggplot(students, aes(x = Height)) +
geom_histogram(binwidth = 2)
```



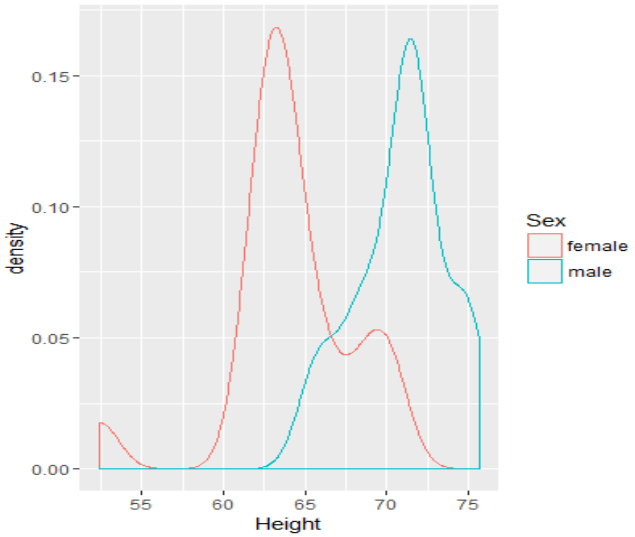

```
# Using Proportions to visualise data
ggplot(students, aes(x = Height)) +
geom_histogram(binwidth = 2, aes(y =
..density..))
```



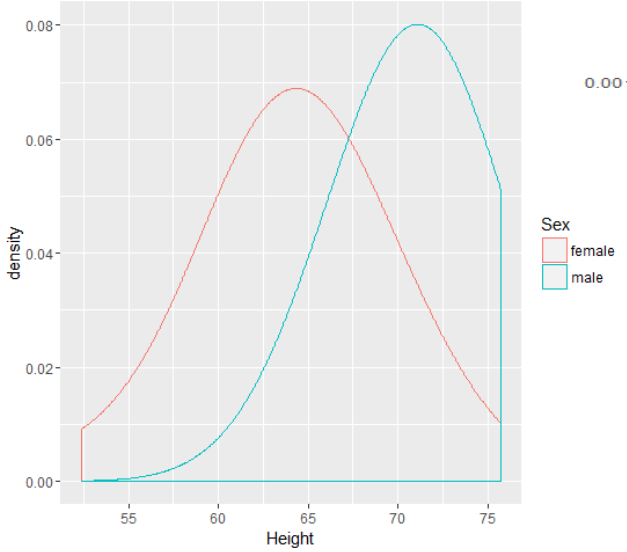
Plotting density plots

- Similar to histograms, but are smooth
- The adjust argument regulates “smoothness”
- Larger values = smoother graphs

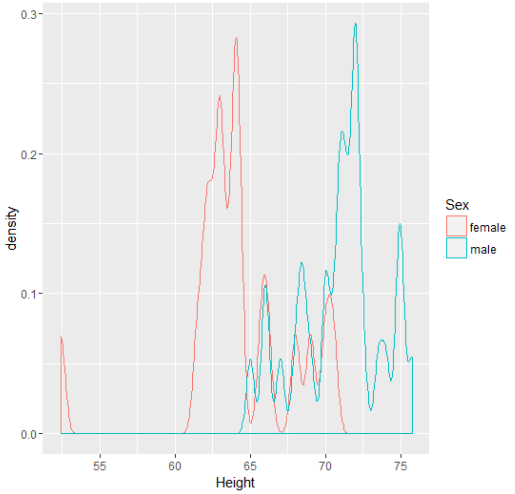
ggplot(students, aes(x = Height)) +
geom_density(aes(group=Sex, colour=Sex))



ggplot(students, aes(x = Height)) +
geom_density(aes(group=Sex, colour=Sex), adjust=4)



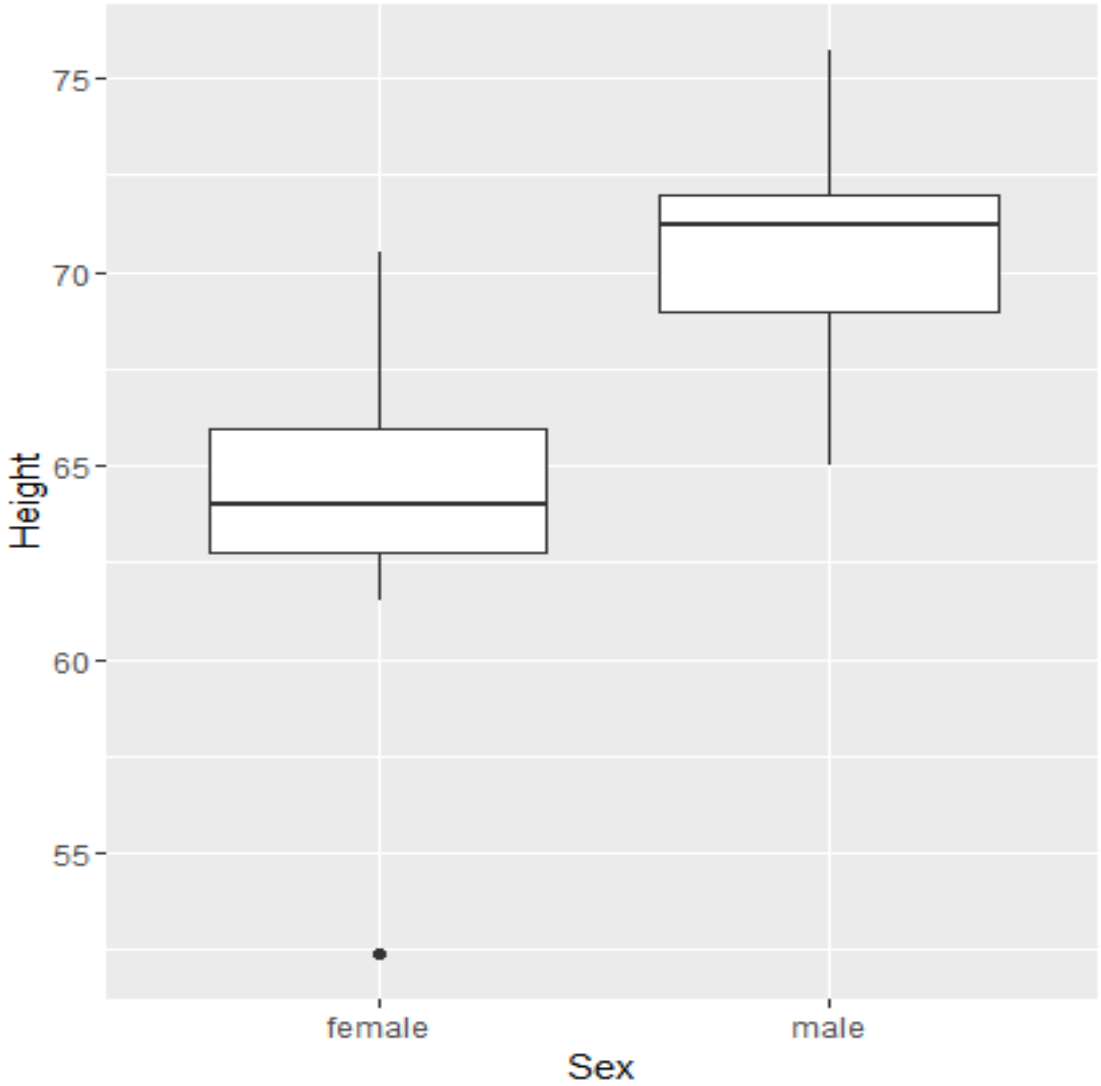
ggplot(students, aes(x = Height)) +
geom_density(aes(group=Sex, colour=Sex), adjust = 0.25)



Boxplots

- Highly Summarised representations
- Provide:
  - Five number summary (min, max, median, 1st and 3rd quartiles)
  - Locations of outliers
- By default it is used to compare two distributions
- But you can force it to handle a single variable

```
ggplot(students, aes(x = Sex, y = Height)) +
geom_boxplot()
```



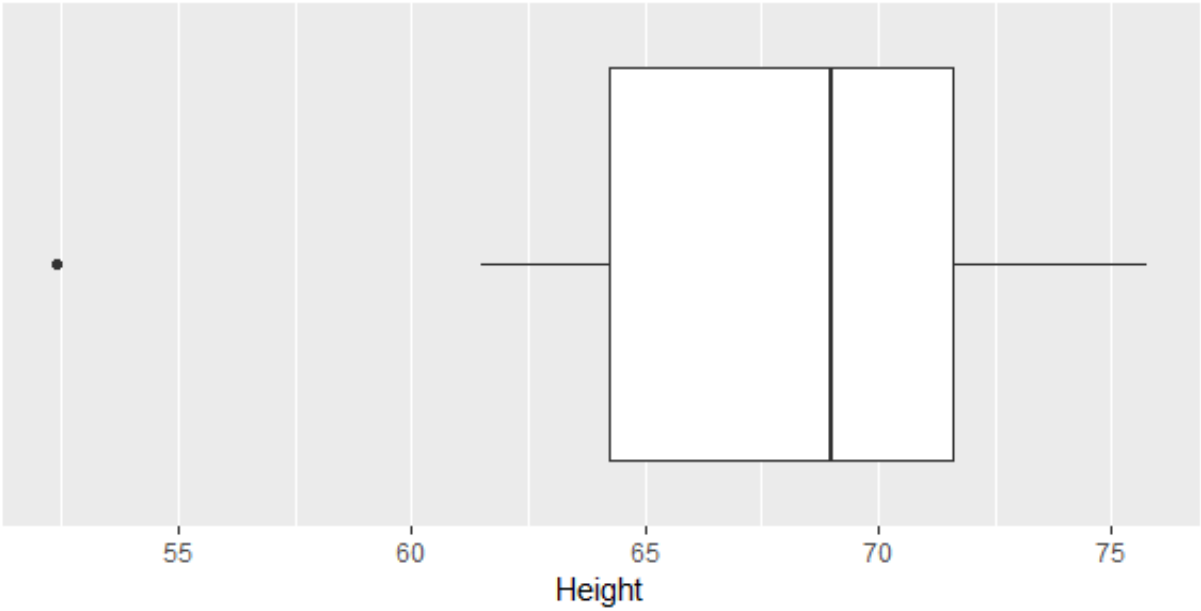
Boxplots for single variables

- Create a fake grouping variable

ggplot(students, aes(x = factor(0), y = Height)) +
geom_boxplot() + xlab("") +
scale_x_discrete(breaks = NULL) + coord_flip()

xlab sets the title of the x-axis

scale_x_discrete: use continuous positions even with a discrete position scale. With this option we are opting not to display breaks

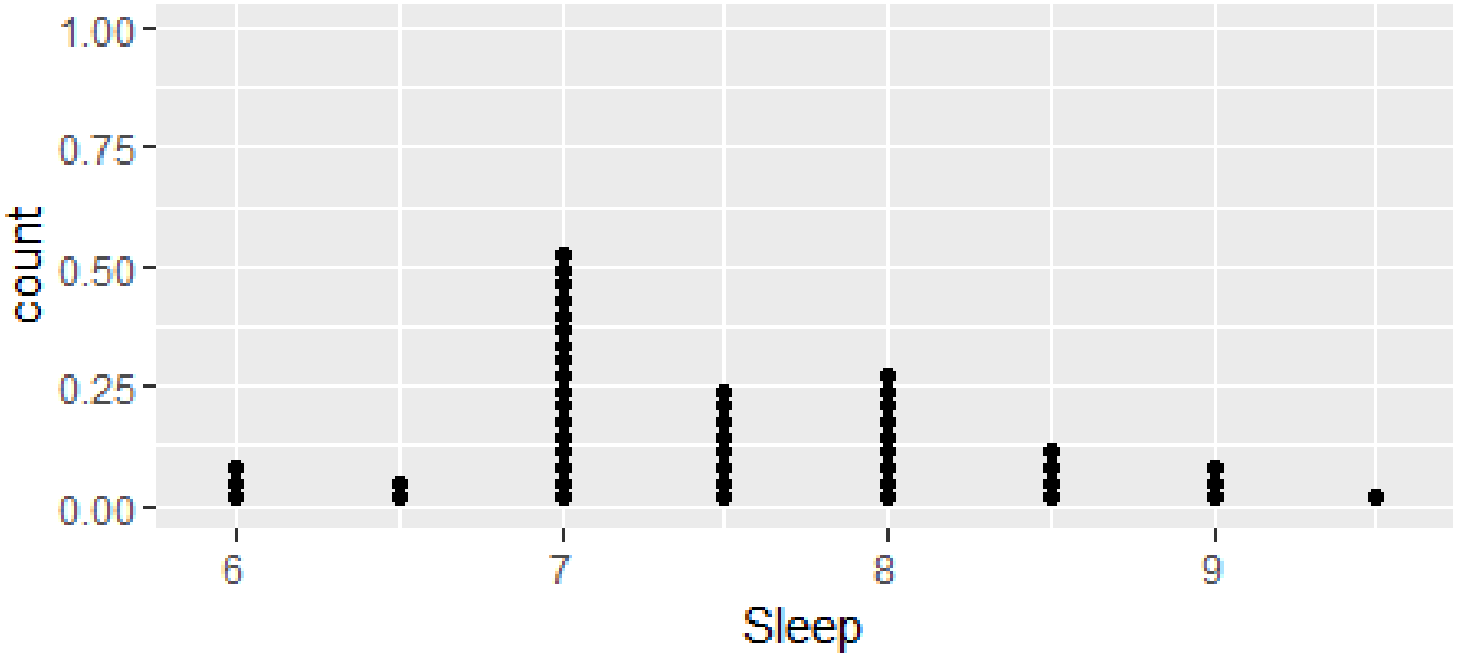


Investigating Relationships between Categorical and Quantitative Variables

- Split a variable among the groups of a quantitative variable to show univariate displays for each group separately

ggplot(students, aes(x=Sleep)) +
geom_dotplot(dotsize = 0.4)

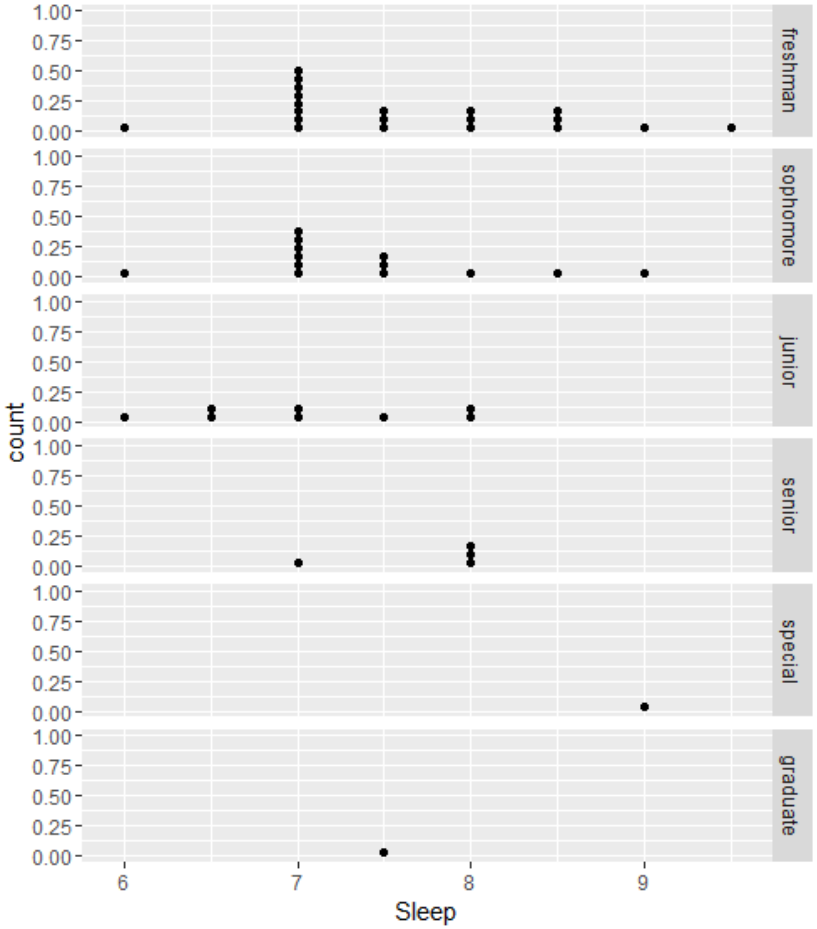
- This is a plot for the whole sample



Investigating Relationships between Categorical and Quantitative Variables

- Using facet_grid(), you can a different row for each plot
- The argument specifies the categorical variable to use
- The period tells ggplot not to split that dimenstion

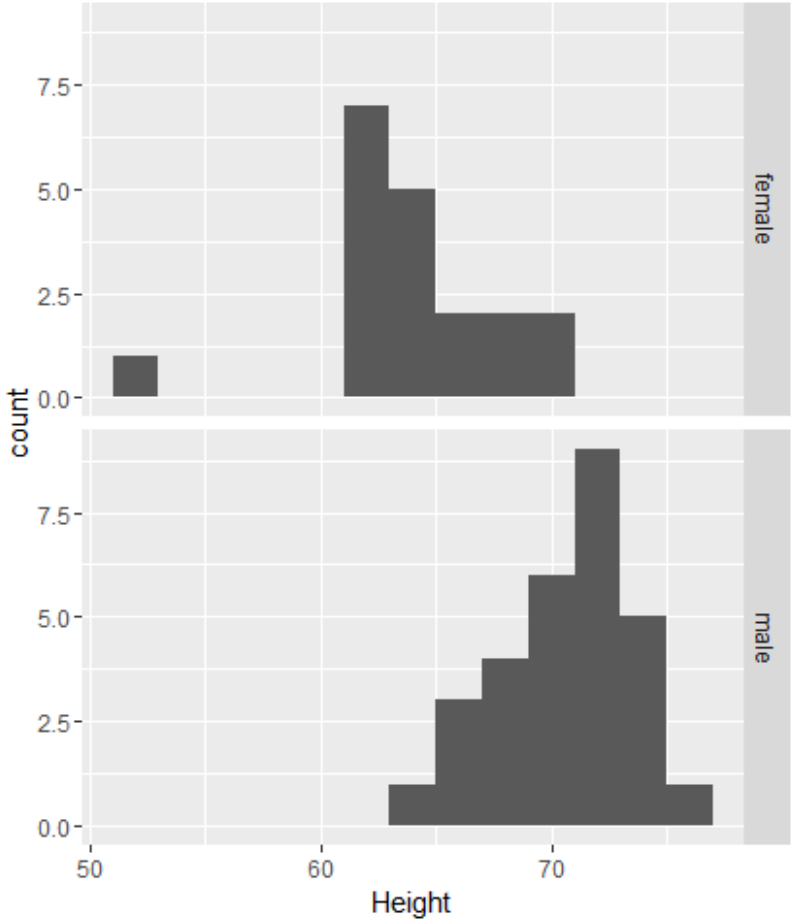
ggplot(students, aes(x=Sleep)) +
geom_dotplot(dotsize = 0.4) + **facet_grid(Level ~ .)**



Investigating Relationships between Categorical and Quantitative Variables

- You can also visualise the data using histograms to compare the different groups

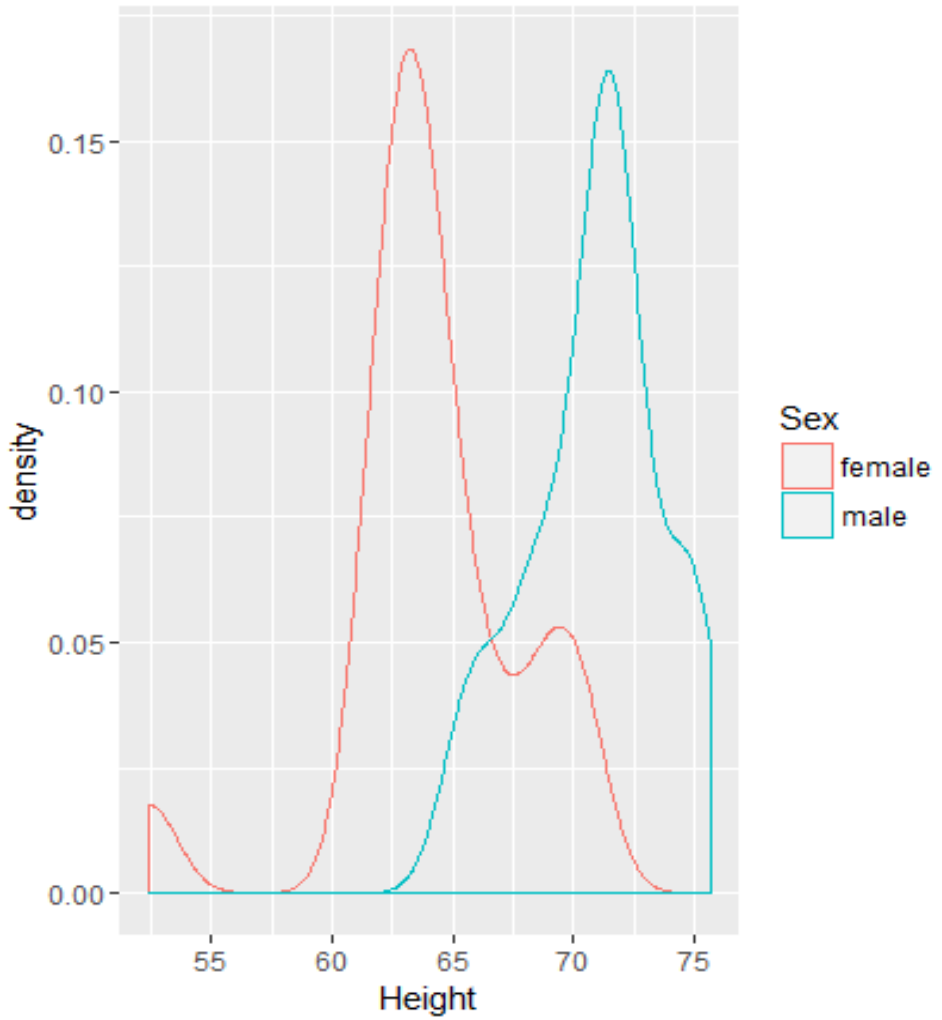
```
ggplot(students, aes(x=Height)) +
geom_histogram(binwidth = 2) + facet_grid(Sex ~ .)
```



Investigating Relationships between Categorical and Quantitative Variables

- Density plots are also useful, especially when coupled with the colour aesthetic to distinguish between the categories

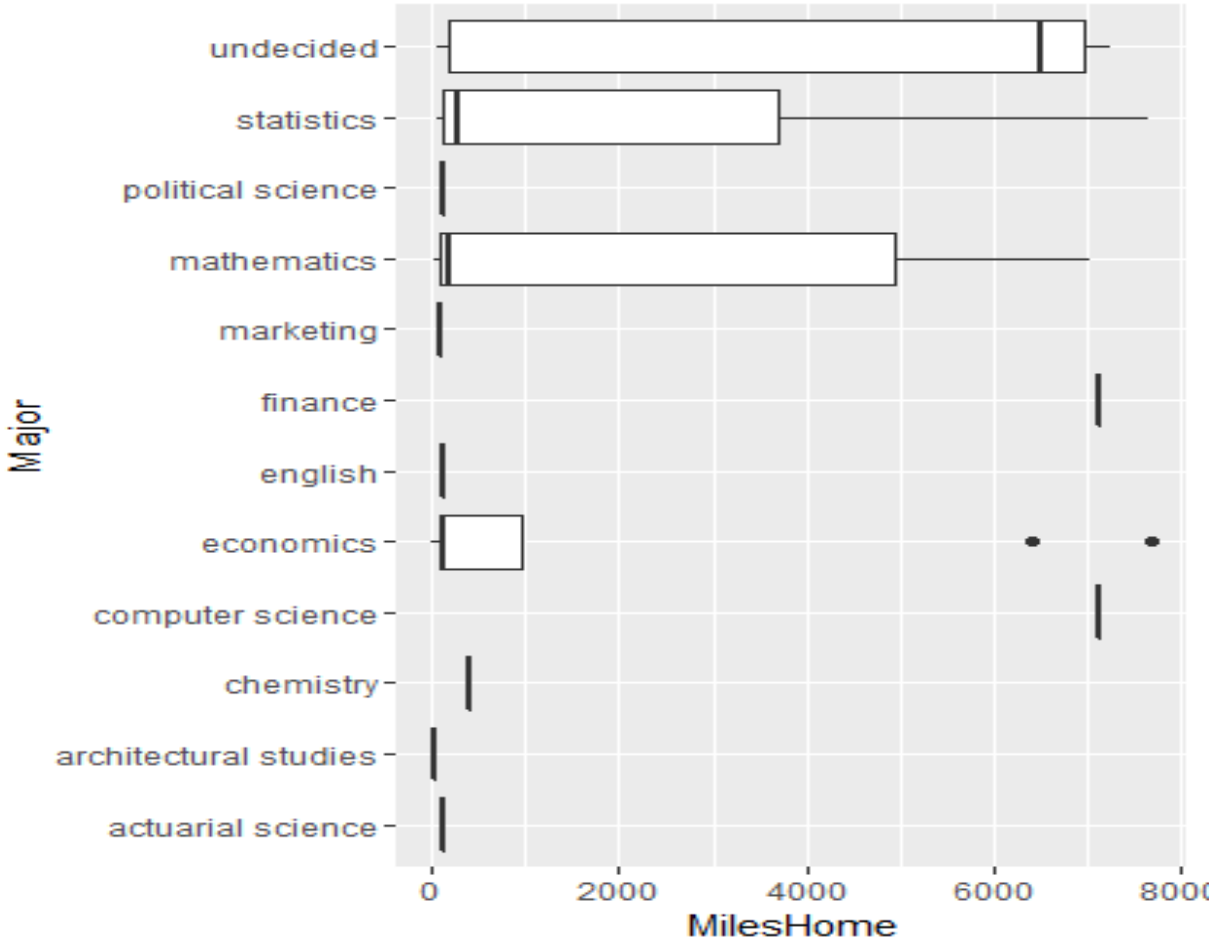
```
ggplot(students, aes(x=Height, color = Sex)) +
geom_density()
```



Investigating Relationships between Categorical and Quantitative Variables

- Side-by-side boxplots are useful if you have many categories to show on a single visualisation

```
ggplot(students, aes(x=Major, y = MilesHome)) +
geom_boxplot() + coord_flip()
```



Download Sleep Study

- Download SleepStudy.csv (using a browser) from

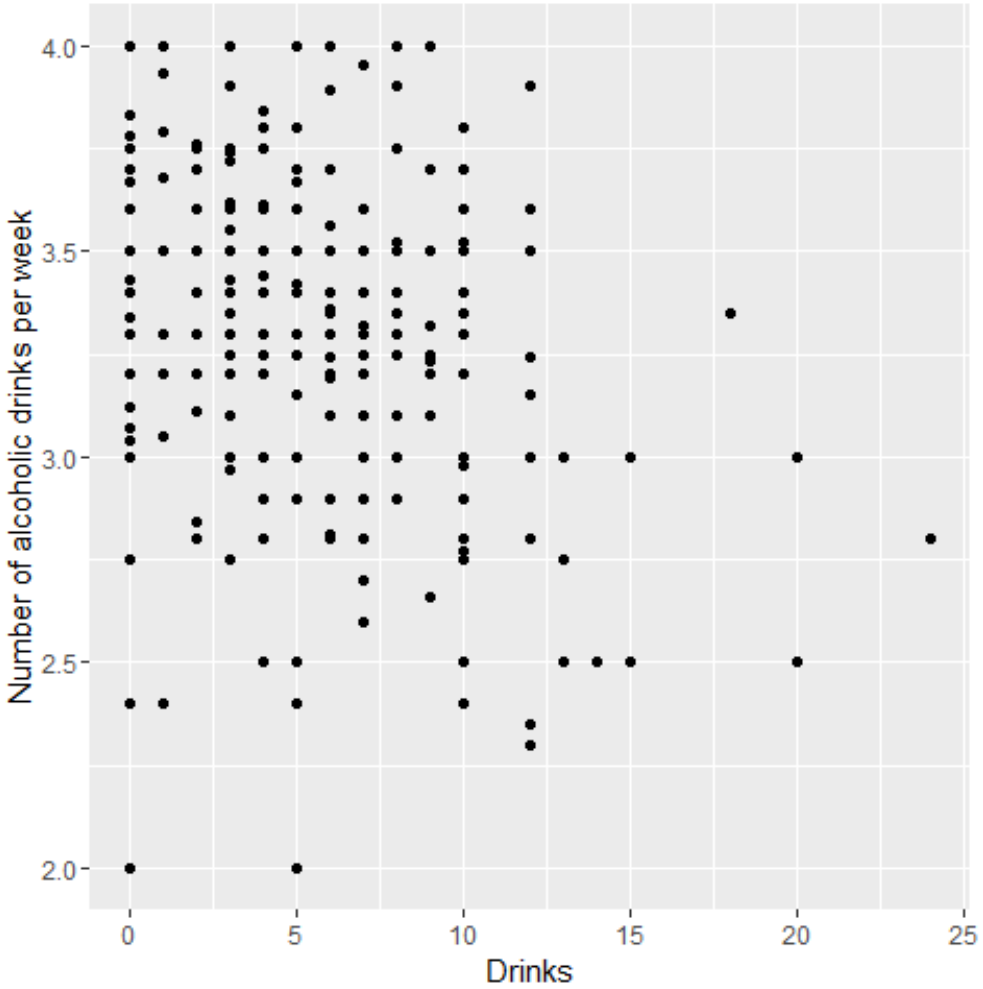
- Load the following datasets:

```
SleepStudy <- read.csv(file.choose()) # SleepStudy.csv
```

Investigating the relationship between two quantitative variables

- The scatterplot is the most useful way of displaying two quantitative variables
- Consider the number of drink consumed by a student and their GPA

```
ggplot(SleepStudy, aes(x=Drinks, y=GPA)) +
        geom_point() +
        ylab("Number of alcoholic drinks per week")
```



Check the correlation between drinks and GPA:

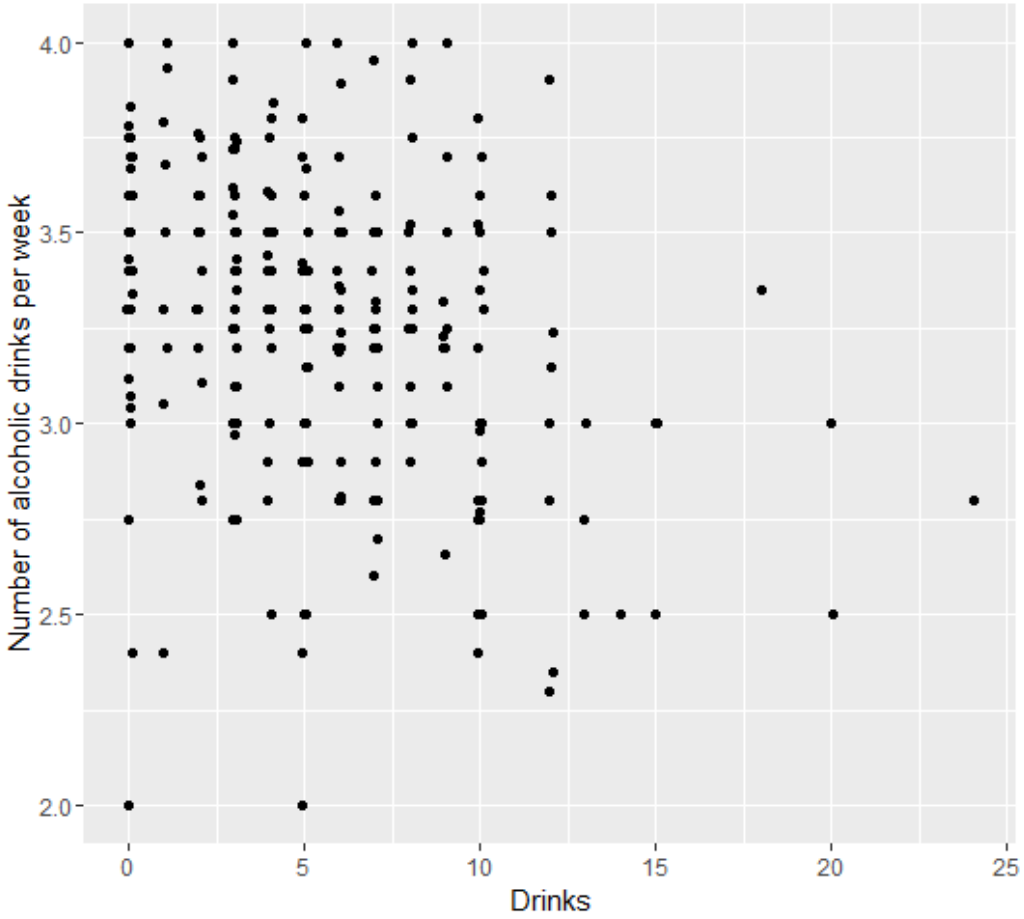
- **with**(SleepStudy, **cor**(Drinks, GPA, method="spearman"))
- **with**(SleepStudy, **cor**(Drinks, GPA, method="kendall"))

Investigating the relationship between two quantitative variables

- It is likely that you have overlapping points in the previous scatterplot
  - Small number of drinks
  - Some students have equal GPA

- Solution: jitter the points along the width of the plot to clarify

```
ggplot(SleepStudy, aes(x=Drinks, y=GPA)) +
        geom_point(position=position_jitter(w=0.2, h=0))
        +  ylab("Number of alcoholic drinks per week")
```



Summary of ggplot geometric objects

- **geom_bar()** – creates a layer with bars representing different statistical properties
- **geom_point()** – creates a layer showing the data points (similar to a scatter plot)
- **geom_line()** – creates a layer that connects data points with a straight line
- **geom_smooth()** – creates a layer that contains a “smoother” – i.e. a line that summarises the data as a whole rather than connecting individual data points

Summary of ggplot geometric objects

- **geom_histogram()** – creates a layer with a histogram on it
- **geom_boxplot()** – creates a layer with a box-whisker diagram
- **geom_text()** – creates a layer with text on it
- **geom_density()** – creates a layer that contains a density plot on it
- **geom_errorbar()** – creates a layer with error bars on it
- **geom_hline()** and **geom_vline()** – creates a layer with a user-defined horizontal or vertical line respectively on it

Lesson 8: Wrap-up

- Advanced ggplot2
  - Used the **with** function
  - Exploited implicit order in the data
  - Plotted categorical and quantitative variables
  - Summarised of ggplot geometric objects

Summary of Training

- Overview of R
- Basic R programming
- Introduction to R's Data types
- Introduction to the Data Frame
- Used dplyr to manipulate Data Frames
- Visualisation using the *base* package
- Advanced visualisations with ggplot2

The End

Thank you for participating