Introduction to Solving Biological Problems Using R - Day 2

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

08/09/2014

Statistics

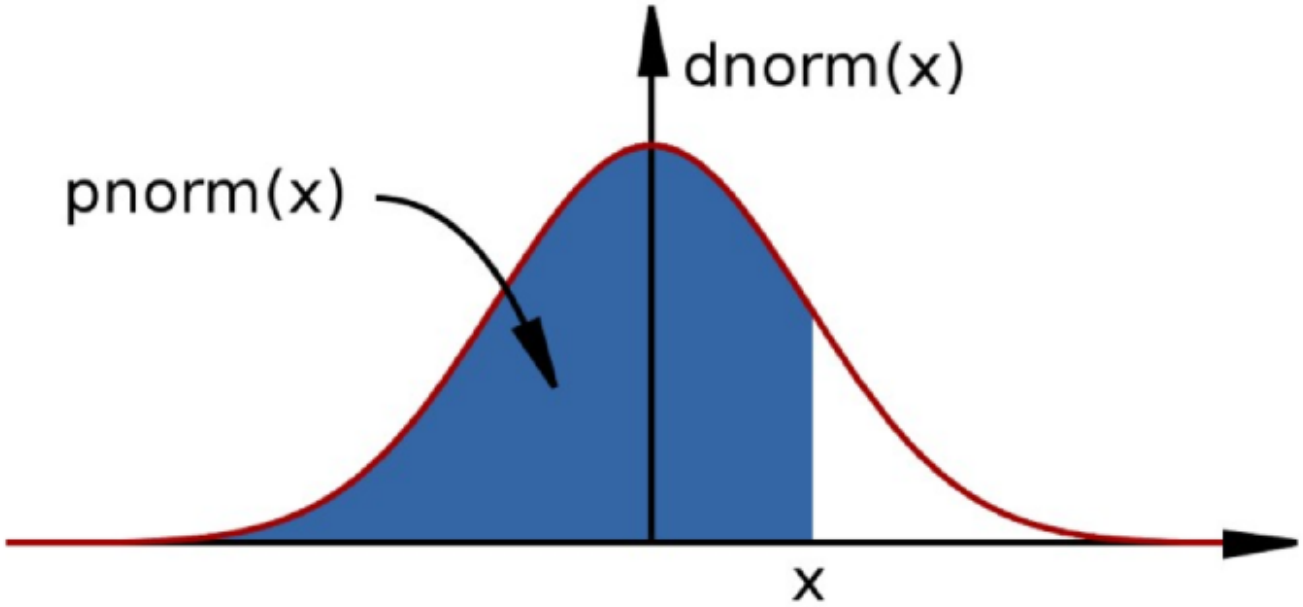
Built-in support for statistics

- R is a statistical programming language
  - Classical statistical tests are built-in
  - Statistical modeling functions are built-in
  - Regression analysis is fully supported
  - Additional mathematical packages are available (`MASS`, Waves, sparse matrices, etc)

Distribution functions

- Most commonly used distributions are built-in, functions have stereotypical names, e.g. for normal distribution
- `pnorm` - cumulative distribution for x
- `qnorm` - inverse of pnorm (from probability gives x)
- `dnorm` - distribution density
- `rnorm` - random number from normal distribution

Distribution functions



- available for variety of distributions: `punif` (uniform), `pbinom` (binomial), `pnbinom` (negative binomial), `ppois` (poisson), `pgeom` (geometric), `phyper` (hyper-geometric), `pt` (T distribution), pf (F distribution)

Distribution functions

- 10 random values from the Normal distribution with mean 10 and standard deviation 5

```
rnorm(10,mean=10,sd=5)
```

```
##  [1] 11.355  7.765  5.746 14.201 14.352  8.618 12.474 16
```

- The probability of drawing 10 from this distribution

```
dnorm(10,mean=10,sd=5)
```

```
## [1] 0.07979
```

```
dnorm(100,mean=10,sd=5)
```

```
## [1] 3.517e-72
```

Distribution functions

- The probability of drawing a value smaller than 10

```
pnorm(10,mean=10,sd=5)
```

```
## [1] 0.5
```

- The inverse of `pnorm`

```
qnorm(0.5,mean=10,sd=5)
```

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

- How many standard deviations for statistical significance?

```
qnorm(0.95,mean=0,sd=1)
```

```
## [1] 1.645
```

Two sample tests: Basic data analysis

- Comparing 2 variances

```
var.test()
```

- Comparing 2 sample means with normal errors

```
t.test()
```

- Comparing 2 means with non-normal errors

```
wilcox.test()
```

- See also `prop.test()`,`cor.test()`,`chisq.test()`,`fisher.test()`

Comparison of 2 data sets example: Basic data analysis

- Men, on average, are taller than women.

1. Determine whether variances in each data series are different.
   - Variance is a measure of sampling dispersion, a first estimate in determining the degree of difference
   - e.g. Fishers' F test
2. Comparison of the mean heights. e.g. Student's t test, Wilcoxon's rank sum test
   - Determine probability that mean heights are really drawn from different sample populations
   - e.g. Student's t-test, Wilcoxon's rank sum test

Comparison of 2 data sets example .Fishers F test

- Read in the data file into a new object, `heightData`

```
heightData<-read.csv("1.5_heightData.csv")
```

- **attach** the data frame so we don't have to refer to it by name all the time

```
attach(heightData)
```

- Do the two sexes have the same variance?

```
var.test(Female,Male)
```

Comparison of 2 data sets example

```
##
##  F test to compare two variances
##
## data:  Female and Male
## F = 1.007, num df = 99, denom df =
## 99, p-value = 0.9714
## alternative hypothesis: true ratio of variances is not e
## 95 percent confidence interval:
##  0.6777 1.4970
## sample estimates:
## ratio of variances
##              1.007
```

Comparison of 2 data sets example. Student' t test

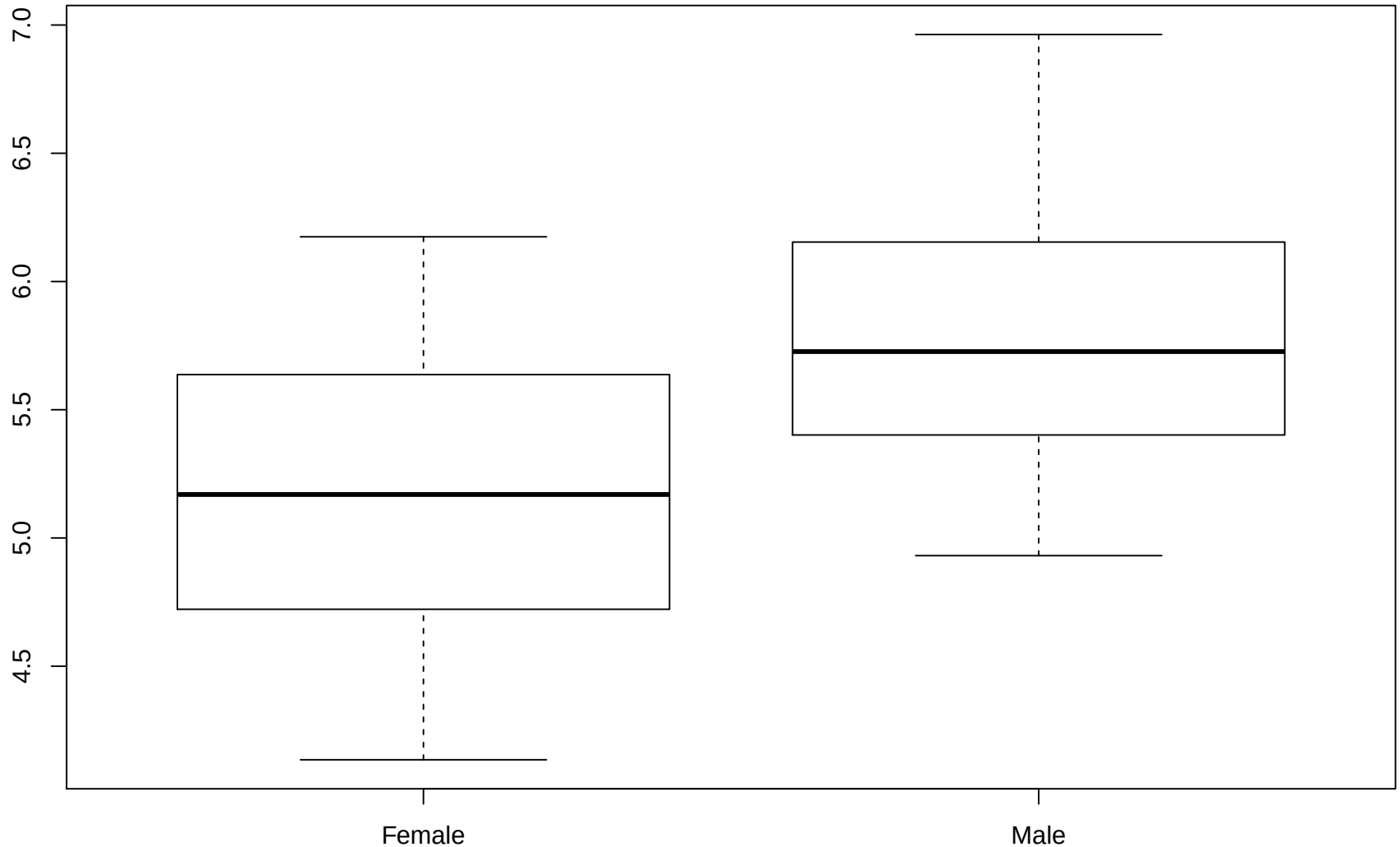
- Student's t test is appropriate for comparing the difference in mean height in our data. We need a one-tailed test

```
t.test(Female,Male, alternative="less")
```

```
##
##  Welch Two Sample t-test
##
## data:  Female and Male
## t = -8.451, df = 198, p-value =
## 3.109e-15
## alternative hypothesis: true difference in means is less
## 95 percent confidence interval:
##    -Inf -0.508
## sample estimates:
## mean of x mean of y
##     5.169     5.800
```

Comparison of 2 data sets example. Review findings

```
boxplot(heightData)
```



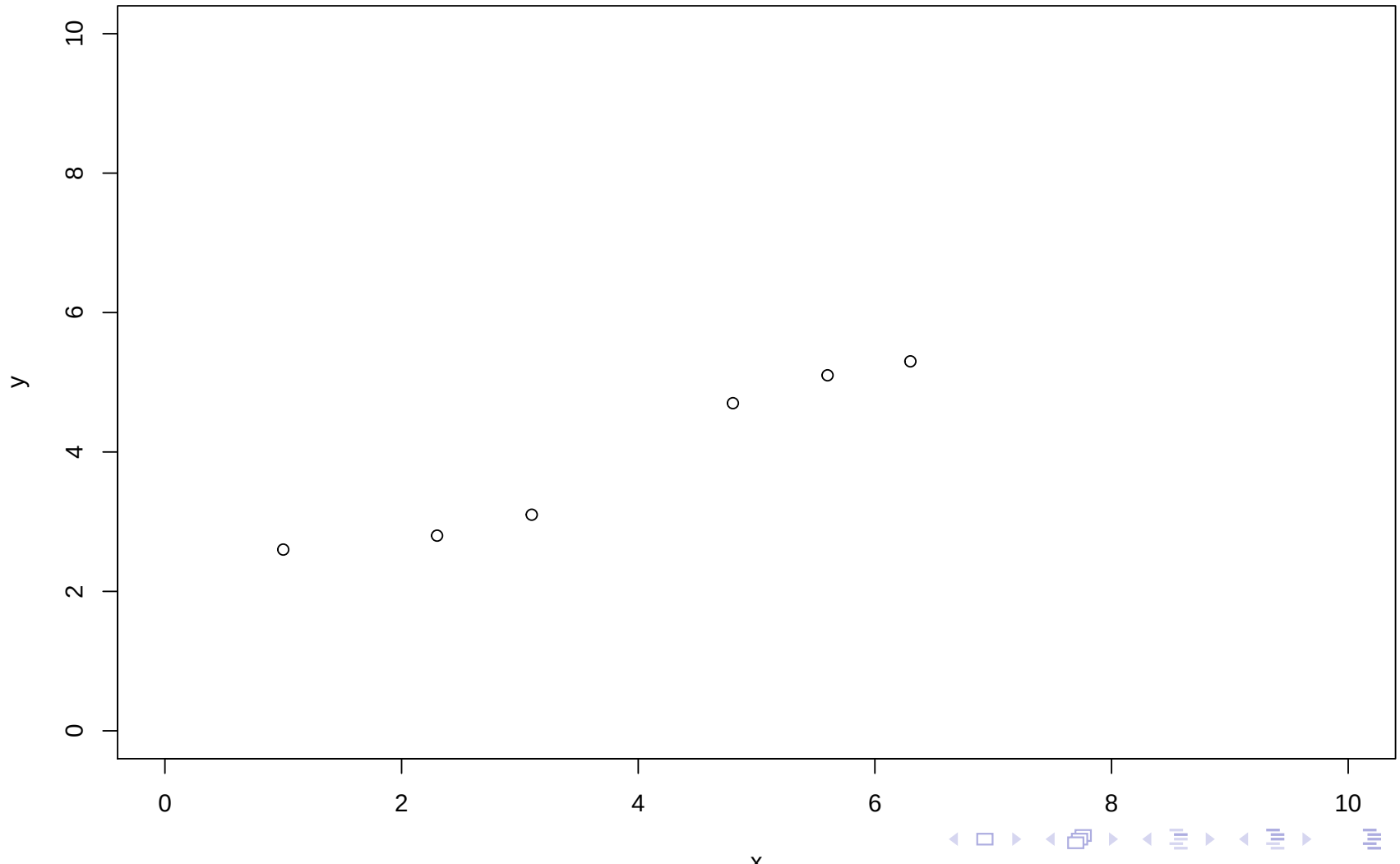
Linear regression: Basic data analysis

- Linear modeling is supported by the function `lm()`
  - `example(lm)` the output assumes you know a fair bit about the subject
- `lm` is really useful for plotting lines of best fit to XY data in order to determine intercept, gradient & Pearson's correlation coefficient
  - This is very easy in R
- Three steps to plotting with a best fit line

1. Plot XY scatter-plot data
2. Fit a linear model
3. Add bestfit line data to plot with `abline()` function

Typical linear regression analysis: Basic data analysis

```
x<-c(1, 2.3, 3.1, 4.8, 5.6, 6.3)
y<-c(2.6, 2.8, 3.1, 4.7, 5.1, 5.3)
plot(y~x, xlim=c(0,10),ylim=c(0,10))
```



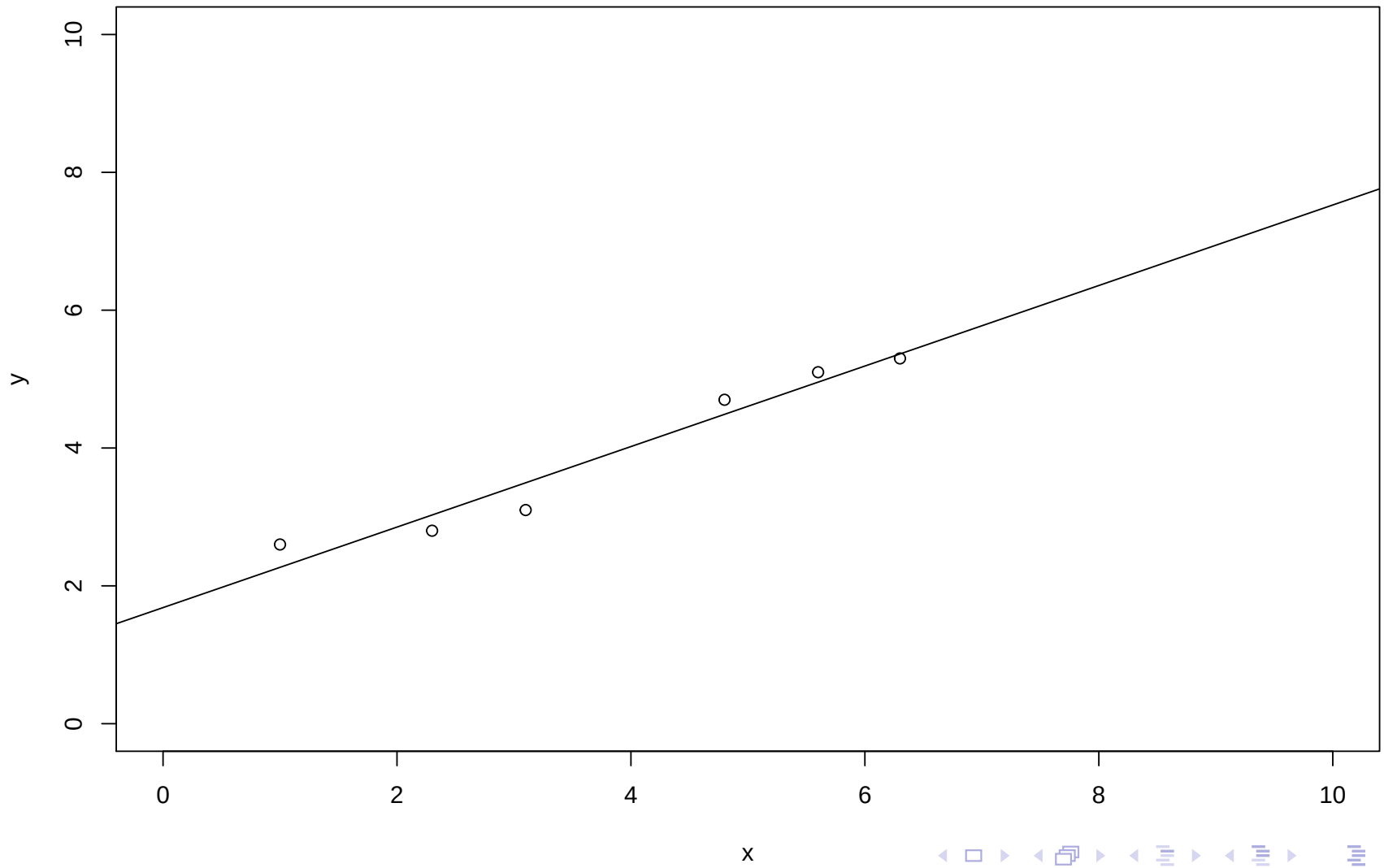
Typical linear regression analysis: Basic data analysis

```
myModel<-lm(y~x)
summary.lm(myModel)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##       1       2       3       4       5
##  0.3316 -0.2278 -0.3952  0.2117  0.1443
##       6
## -0.0646
##
## Coefficients:
##             Estimate Std. Error t value
## (Intercept)   1.6842     0.2906    5.80
## x             0.5842     0.0679    8.61
##             Pr(>|t|)
```

Typical linear regression analysis: Basic data analysis

```
plot(y~x, xlim=c(0,10),ylim=c(0,10))
abline(myModel)
```



Modelling formulae

- R has a very powerful formula syntax for describing statistical models
- Suppose we had two explanatory variables `x` and `z` and one response variable `y`
- We can describe a relationship between, say, `y` and `x` using a tilde `~`, placing the response variable on the left of the tilde and the explanatory variables on the right:
  - `y~x`
- It is very easy to extend this syntax to do multiple regressions, ANOVAs, to include interactions, and to do many other common modelling tasks. For example

Modelling formulae

```
y~x #If x is continuous this is linear regression
y~x #If x is categorical, this is ANOVA
y~x+z #If x and z are continuous, this is multiple regress
y~x+z #If x and z are categorical, this is two-way ANOVA
y~x+z+x:z # : is the symbol for the interaction term
y~x*z     # * is a shorthand for x+z+x:z
```

Summary of the linear model object: summary(myModel)

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##       1       2       3       4       5       6
##  0.3316 -0.2278 -0.3952  0.2117  0.1443 -0.0646
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.6842     0.2906    5.80   0.0044 **
## x             0.5842     0.0679    8.61   0.0010 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
##
## Residual standard error: 0.311 on 4 degrees of freedom
## Multiple R-squared:  0.949,  Adjusted R-squared:  0.936
## F-statistic: 74.1 on 1 and 4 DF,  p-value: 0.001
```

Exercise: The coin toss

To learn how the distribution functions work, try simulating tossing a fair coin 100 times and then show that it is fair

- We can model a coin toss using the binomial distribution. Use the `rbinom` function to generate a sample of 100 coin tosses. Look up the binomial distribution help page to find out what arguments this function needs
- How many heads or tails were there in your sample? You can do this in two ways; either select the number of successes using indices, or convert your sample to a factor and get a summary of the factor
- If we toss a coin 50 times, what is the probability that we get exactly 25 heads? What about 25 heads or less? Use `dbinom` and `pbinom` to find out
- The argument to `dbinom` is a vector, so try calculating the probabilities for getting any number of coin tosses from 0 to 50 in fifty trials using `dbinom`. Plot these probabilities using `plot`. Does this plot remind you of anything?

Coin toss answers

- To simulate a coin toss, give `rbinom` a number of observations, the number of trials for each observation, and a probability of success

```
coin.toss<-rbinom(100, 1, 0.5)
```

- Because we only specified one trial per observation, we either have an outcome of 0 or 1 successes. To get the number of successes, use indices or a factor to look up the number of 1s in the coin.toss vector (your numbers will vary)

```
length(coin.toss[coin.toss==1])
```

```
## [1] 50
```

```
summary(factor(coin.toss))
```

```
##  0  1
```

Coin toss answers

- The probability of getting exactly 25 heads from 50 observations of a fair coin

```
dbinom(25, 50, 0.5)
```

- The probability of getting 25 heads or less from 50 observations of a fair coin

```
pbinom(25, 50, 0.5)
```

Coin toss answers

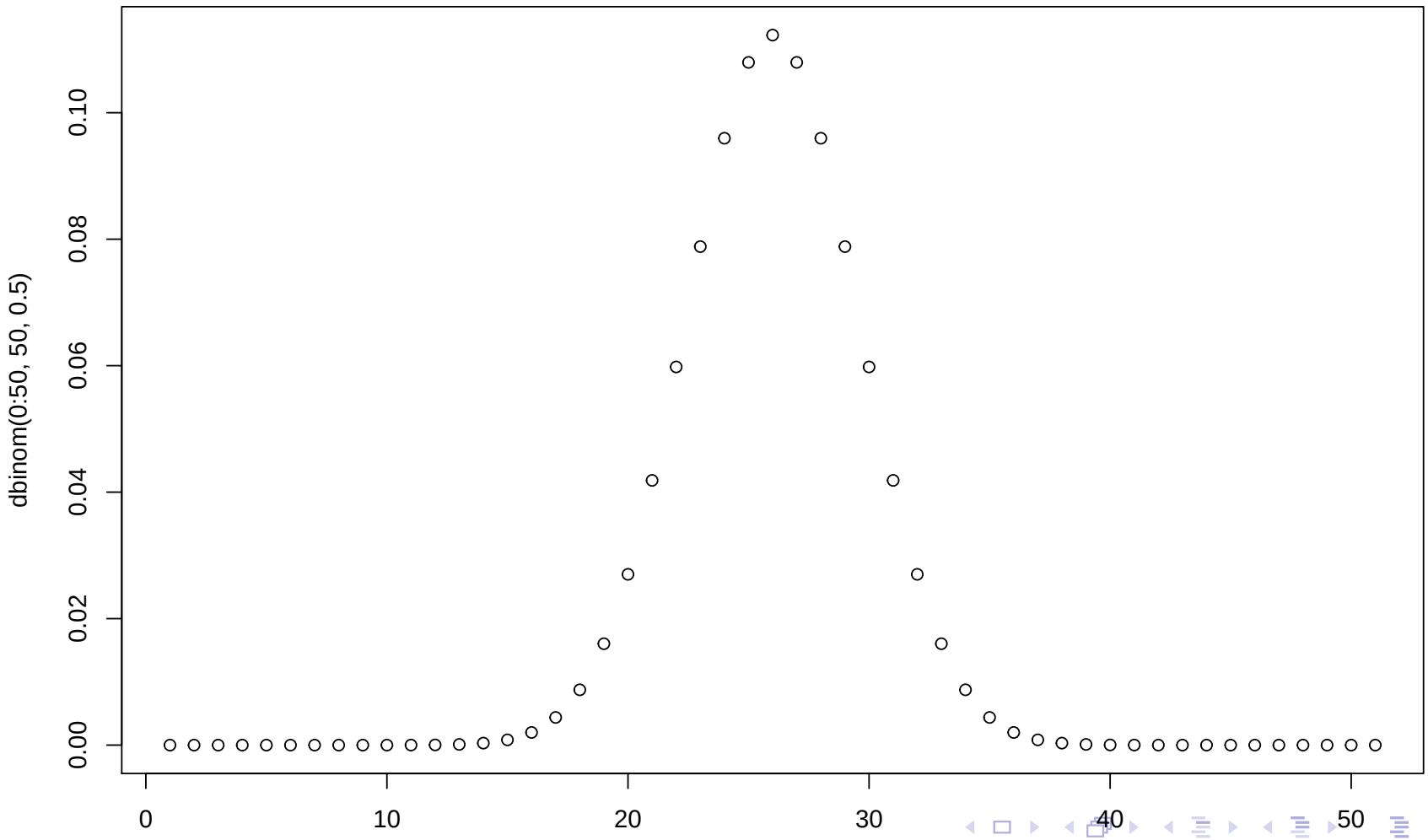
- The probabilities for getting all numbers of coin tosses from 0 to 50 in fifty trials

```
dbinom(0:50, 50, 0.5)
```

Coin toss answers

- To plot this distribution, which should resemble a normal distribution

```
plot(dbinom(0:50, 50, 0.5))
```



Exercise: Linear modelling example

Mice have varying numbers of babies in each litter. Does the size of the litter affect the average brain weight of the offspring? We can use linear modelling to find out. (This example is taken from John Maindonald and John Braun's book Data Analysis and Graphics Using R (CUP, 2003), p140-143.)

- Install and load the `DAAG` package. The `litters` data frame is part of this package. Take a look at it. How many variables and observations does it have? Does `summary` tell you anything useful? What about `plot`?
- Are any of the variables correlated? Look up the `cor.test` function and use it to test for relationships.

Exercise: Linear modelling example

- Use `lm` to calculate the regression of brain weight on litter size, brain weight on body weight, and brain weight on litter size and body weight together
- Look at the coefficients in your models. How is brain weight related to litter size on its own? What about in the multiple regression? How would you interpret this result?

Linear modelling answers

- To install and load the package and look at `litters`

```
install.packages("DAAG")
library(DAAG)
litters
summary(litters)
plot(litters)
```

- To calculate correlations between variables

```
attach(litters)
cor.test(brainwt, lsize)
cor.test(bodywt, lsize)
cor.test(brainwt, bodywt)
```

Linear modelling answers

- To calculate the linear models

```
## Loading required package: lattice
```

```
lm(brainwt~lsize)
```

```
##
## Call:
## lm(formula = brainwt ~ lsize)
##
## Coefficients:
## (Intercept)        lsize
##     0.44700     -0.00403
```

Linear modelling answers

```
lm(brainwt~bodywt)
```

```
##
## Call:
## lm(formula = brainwt ~ bodywt)
##
## Coefficients:
## (Intercept)       bodywt
##      0.3355       0.0105
```

Linear modelling answers

```
lm(formula = brainwt ~ lsize + bodywt)
```

```
##
## Call:
## lm(formula = brainwt ~ lsize + bodywt)
##
## Coefficients:
## (Intercept)         lsize        bodywt
##     0.17825       0.00669       0.02431
```

- *Interpretation*: brain weight decreases as litter size increases, but brain weight increases proportional to body weight (when bodywt is held constant, the lsize coefficient is positive: 0.00669). This is called 'brain sparing'; although the offspring get smaller as litter size increases, the brain does not shrink as much as the body

Writing custom scripts for data analysis

The R scripting language

- A script is a series of instructions that when executed sequentially automates a task
  - A script is a good solution to a repetitive problem
- The art of good script writing is
  - understanding exactly what you want to do
  - expressing the steps as concisely as possible
  - making use of error checking
  - including descriptive comments

The R scripting language

- R is a powerful scripting language, and embodies aspects found in most standard programming environments
  - procedrual statements
  - loops
  - functions
  - conditional branching
- Scripts may be written in any standard text editor, e.g. notepad, gedit, kate
  - we will use RStudio

Colony forming experiment

- We have been asked y some collaborators to analyse some trial data to see if an experiment will work
- We are interested in the behaviour of a gene, X, which is involved in a cell proliferation pathway
- This pathway causes cells to proliferate in the presence of a compound, Z
- Gene X turns the pathway off, reducing cell proliferation
- Our collaborators want to test what happens when we knock down X in the presence of Z
- To do this, they want to grow cell colonies in the presence of Z, with or without X, and count the number of colonies that result

Initial Trial

- Our collaborators have sent us a first batch of test data, growing colonies in different concentrations of compound Z, and replicating each Z concentration three times
- Does increasing concentration of Z have an effect on colony growth?
- We want to do the following:
  - Load the data into R
  - Plot the data to inspect them
  - Calculate an Analysis of Variance to see if the growth is influenced by Z concentration
  - Calculate the mean growth for each level of Z concentration, to see the direction of change

Initial trial exercise

- The trial data are in the file '2.1_colony_trial.csv'. Load this file into R using the command we learnt yesterday
- Plot the data using a formula, to see how Z affects colony Count. Recall how we did this earlier with linear modelling, with independant variable x and dependent variable y: `plot(y~x)`
- Calculate an analysis of variance for the data. The R function for ANOVA is `aov`, which works like `lm()` from earlier
- There are four concentrations of Z, and each concentration has been replicated three times. What is the mean colony count for each concentration? See if you can figure out a way to calculate this with what we learnt yesterday. You will need to use logical indexing and you may want to use a loop

Walkthrough - Importing the data

- Use `read.csv` to load the data

```
colony <- read.csv("2.1_colony_trial.csv")
```

```
##          Z Replicate Count
## 1     None         1   150
## 2     None         2   188
## 3     None         3   223
## 4      Low         1    87
## 5      Low         2    40
## 6      Low         3    53
## 7   Medium         1     5
## 8   Medium         2     1
## 9   Medium         3     9
## 10    High         1     0
## 11    High         2     0
## 12    High         3     0
```

Walkthrough - Importing the data

```
##      Z Replicate Count
## 1 None         1   150
## 2 None         2   188
## 3 None         3   223
## 4  Low         1    87
```

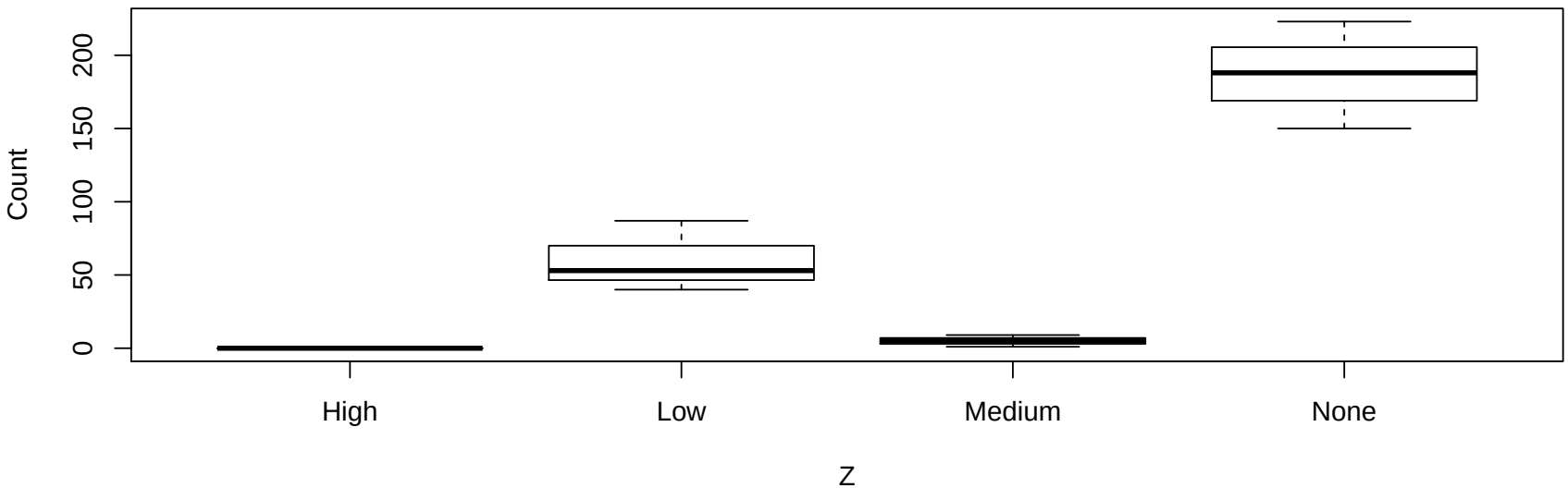
- The data frame has three columns; Z, Replicate and Count. We want to know how Z affects the number of colonies (Count). To do this, we need to summarise the data over all replicates for each concentration of Z.
- We will attach the data frame to our workspace, so we can refer to the variables without referring to the data frame all the time

```
attach(colony)
```

Walkthrough - Plotting

- We want to plot the colony growth in response to changing Z concentration.
- Z is the explanatory variable and Count is the response variable

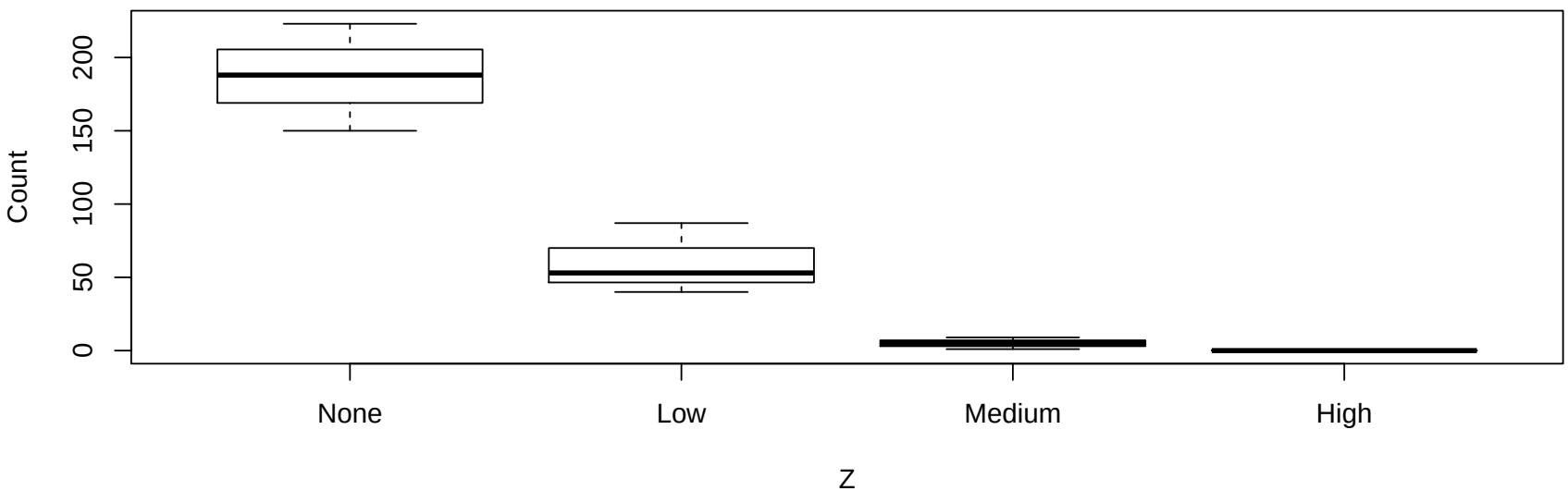
```
plot(Count ~ Z)
```



Walkthrough - Plotting

- We can improve on this. Firstly, we want to order the Z categories. Z is a factor, so we need to supply new levels to this factor in the colony data frame

```
Z  <- factor(Z, levels=c("None", "Low",
                         "Medium", "High"))
plot(Count ~ Z)
```



Walkthrough - Analysis of Variance

- We can use the same formula syntax to calculate an analysis of variance:

```
colony.aov <- aov(Count~Z)
summary(colony.aov)
```

```
##             Df Sum Sq Mean Sq F value Pr(>F)
## Z            3  68154   22718    46.9  2e-05 ***
## Residuals    8   3876     484
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
```

This tells us what we can already see from the plot, that there is a highly significant relationship between Z concentration and colony growth

Calculating group means

- We can calculate a mean for a particular group like this:

```
mean(colony[Z == "None",]$Count)
```

```
## [1] 187
```

```
mean(colony[Z == "Low",]$Count)
```

```
## [1] 60
```

```
mean(colony[Z == "Medium",]$Count)
```

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

```
mean(colony[Z == "High",]$Count)
```

```
## [1] 0
```

Calculating group means

- We could generalise this with a for loop:

```
for (z in levels(Z)){
  print(mean(colony[Z==z,]$Count))
}
```

```
## [1] 187
## [1] 60
## [1] 5
## [1] 0
```

- But there is a better way

The tapply function

- The apply family of functions allow us to group data by variable and calculate something for each group
- Assume we have the following data for heights of 5 males and females

```
data <- data.frame(gender = c("M","M","F","F","F"),
                   height=c(6,6.1,5.8,6,5.95))
data
```

```
##   gender height
## 1      M   6.00
## 2      M   6.10
## 3      F   5.80
## 4      F   6.00
## 5      F   5.95
```

- How can we get the mean of males and females separately?

The tapply function

- The `tapply` function lets us do exactly this

```
tapply( data, groups, function)
```

- in our case:

```
tapply(data$height, data$gender, mean)
```

```
##     F     M
## 5.917 6.050
```

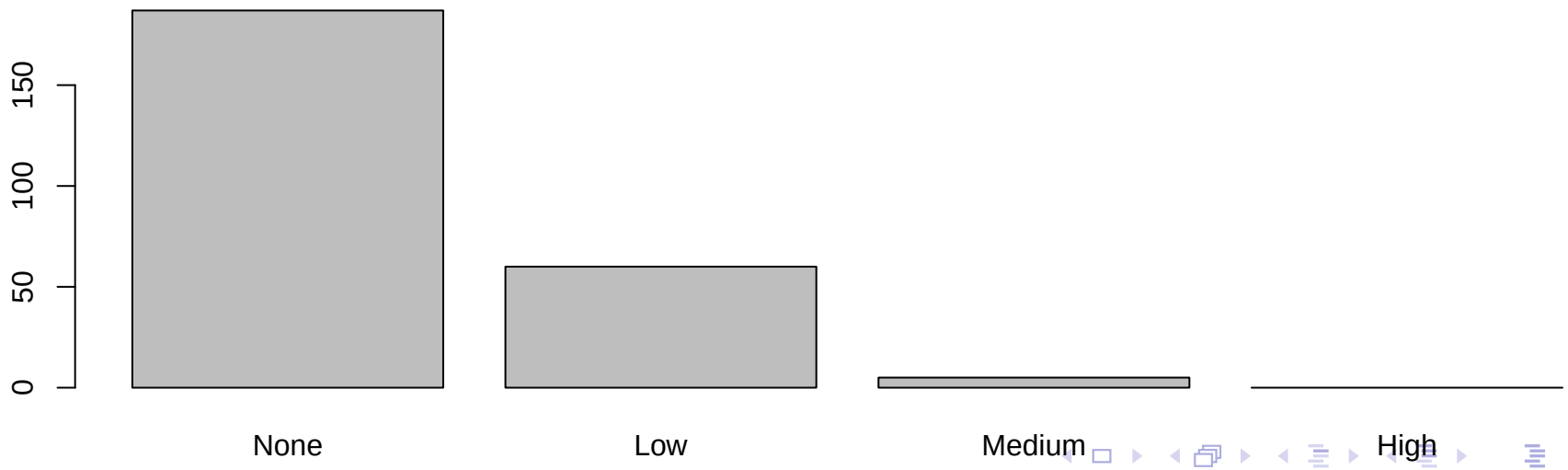
Using tapply on colony

- We can use tapply to calculate group means on colony like this:

```
colony.means <- tapply(Count, Z, mean)
colony.means
```

```
##    None     Low Medium    High
##     187      60       5       0
```

```
barplot(colony.means)
```



A complete script

- We now have a complete script to analyse these data

```
#Load data, order Z and plot
colony <- read.csv("2.1_colony_trial.csv")
colony$Z <- factor(colony$Z, c("None","Low",
                               "Medium","High"))
attach(colony)
plot(Count ~ Z)
#Analysis of variance
colony.aov <- aov(Count~Z)
print(summary(colony.aov))
#Calculate group means
colony.means <- tapply(Count,Z,mean)
print(colony.means)
barplot(colony.means)
detach(colony)
```

- 2.1_colony_1.R)

User functions

Introducing user functions

- All R commands are function calls
- Functions take some input, perform calculations on that input, and return some output
- e.g. `sqrt` is a function that takes a value, calculates the square root of the value, and returns the square root
- `aov` takes a formula referring to some data, calculate the analysis of variance for that data, and returns the model it calculated
- We can define our own functions. User functions extend the capabilities of R by adapting or creating new tasks that are tailored to your specific requirements
- User functions are objects, just like vectors and data frames. This has a few useful implications

Defining a new function

- A function has a name, arguments, procedural steps, and a return value

```
sqXplusX <- function(x){
  x^2 +x
}
```

- `sqXplusX` is the function name
- `x` is the single argument to this function and it exists only within the function
- everything between the brackets `{}` are procedural steps
- the last calculated value is the function return value. We can call `return` explictly

```
sqXplusX(10)
```

```
## [1] 110
```

Named and default arguments

- We can generalise our function by adding a second argument

```
powXplusX <- function(x, power=2){
  x^power + x
}
```

- The `power` argument has a default value of 2; if we don't supply a power when we call the function, `x` will be squared
- Arguments without a default value are required, those with default values are optional

```
powXplusX(10,3) #arguments are matched based on position
```

```
## [1] 1010
```

```
powXplusX(x=10,power=3) # arguments matched based on name
```

```
## [1] 1010
```

Calculation with user functions

- User functions can be used wherever a built-in function can be used:

```
#make some example data
a <- matrix(1:100, ncol=10, byrow=TRUE)
sqXplusX(a)
```

- Functions are R objects, just like a vector or a data frame, and exist in our workspace

```
sqXplusX
```

```
## function(x){
##   x^2 +x
## }
```

Variable scope

- Objects created in functions are not available to the global environment unless returned. They are limited to the scope of the function

```
addone <- function(x) {x <- x+1;x}
x <-1
addone(x)
```

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

```
x
```

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

Variable scope

- The `x` in the global environment has nothing to do with the `x` declared in the function, and is unchanged by the call to the function. To update the global `x`, we would need to assign the return value of the function

```
x <- addone(x)
```

- A function can only return one object, but that object can be a list, so if you have many objects to return, package them up into a list first

Script / function tips

- If your script repeats the same command with different values more than twice, you should consider writing a function to generalise that command
- Writing functions makes your code more easily understandable because they encapuslate a procedure into a well-defined boundary with consistent input/output
- Functions should only do one thing. If a function is doing multiple tasks, try to split it up into multiple functions. This rule of thumb means functions tend to be short, not more than around one or two screens of code
- Look at other functions to get ideas for how to write your own..
  - Display function code by entering the functions' name without brackets ()

Checking input and reporting errors

- A function should fail gracefully if it does not receive valid input when it is called. We can use `if` statements to check for appropriate input
- R has two useful commands to tell the user something is wrong. `warning` prints a message and continues to run the function. `stop` ends the function after printing the message.
- For example, we might rewrite our `powXplusX` function to check that the power argument is a whole number:

```
powXplusX <- function(x, power=2){
  if(power %% 1 != 0) stop ("Power should be a whole numbe
  x^power+x
}
```

Checking input and reporting errors

```
powXplusX(10,3)
```

```
## [1] 1010
```

```
powXplusX(10,3.5)
```

```
## Error in powXplusX(10, 3.5) : Power should be a whole nu
```

Checking input and reporting errors

- R has a very useful set of functions called the `is` family, which check the type of input values. For example:

```
sqXplusX <- function(x){
  if (is.numeric(x)){
    x^2 + x
  } else {
    stop("Input should be numeric")
  }
}
```

```
sqXplusX(10)
```

```
## [1] 110
```

Checking input and reporting errors

```
sqXplusX("ten")
```

```
## Error in sqXplusX("ten") : Input should be numeric
```

The `is.` family consists of; `is.character`, `is.null`,`is.na` `is.data.frame` etc. For full list, type the following and press `TAB`

```
is.
```

Checking input and reporting errors

- Here's another, more concise way to do the same thing:

```
sqXplusX <- function(x){
  if(!is.numeric(x)) stop("Input should be numeric")
  x^2 + x
}
```

- This is not only shorter, but it also gets all the error checking out of the way before the main processing stesp
- You may also find the `%in%` command useful, which checks to see if the elements of one vector are present in another

Checking input and reporting errors

```
levels(colony$Z)
```

```
## [1] "High"   "Low"    "Medium" "None"
```

```
"Low" %in% colony$Z
```

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

```
"Zero" %in% colony$Z
```

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

```
c("None", "Low") %in% colony$Z
```

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

Temperature conversion exercise

- Centigrade to Fahrenheit conversion is given by F $= 9/5$ *C $+32$
- Write a function that converts between temperatures
- The function should take two named arguments
  - temperature `t` is numeric
  - units `unit` is character
- Both arguments should have appropriate default values
- The function should report an appropriate error if inappropriate values are given
  - `if(!is.numeric(t)) { ...}`
  - `if(!(unit %in% c("c","f"))) {...}`
- The function should print out the temperature in Fahrenheit if given in Centigrade, and vice-versa

Building the solution

- It is difficult to write large chunks of code. Instead, start with something that works and build upon it
- e.g. to solve the temperature conversion exercise:
  - write a skeleton function definition (e.g. just a name and brackets)
  - add appropriate argument names and defaults
  - write code to convert Centigrade to Fahrenheit and check it works
  - write code to convert Fahrenheit to Centigrade and check it works
  - add error-checking code, including the checks from the previous slide, and any others you can think of
  - write a set of test calls to confirm that your function handles correct and incorrect input
- If you get stuck, call us for help

Temperature conversion exercise

```
convTemp <- function(t=0, unit="c"){
    if(!is.numeric(t)) stop("Non numeric
                                temperature entered")
    if(!unit %in% c("c","f")){
    stop("Unrecognised temperature unit")
  }
  converted <- 0
  # conversion for centigrade
  if(unit =="c"){
    converted <- 9/5 * t +32
  }
  #conversion for Fahrenheit
  if(unit =="f"){
    converted <- 5/9 * (t-32)
  }
  converted
}
```

Advanced data processing

Combining data from multiple sources

- R has powerful functions to combine heterogeneous data into a single data set
- Gene clustering example data
  - five sets of differentially expressed genes from various experimental conditions
  - file with names of experimentally-verified genes
- Gene clustering exercise
  - combine this dataset into a single table and cluster to see which conditions are similar
  - repeat the clustering but only on a subset of experimentally-verified genes

Combining gene tables

- input files have two columns: gene names and fold change
- we want to combine all five tables into a single table, with 0 for missing values

LpR2	3.5795
fs(1)h	3.1376
CG6954	2.7492
Psa	2.7012
zfh2	2.6247
Fur1	2.4413
ct	2.3804
S	2.3674
rux	2.3574
RhoBTB	2.26
CG14889	2.1735
oc	2.1421
pros	2.0882
Kr-h1	-2.0447
CG5149	-2.1521
tna	-2.2102
CG14888	-2.4346
CG31368	-2.4793
Trim9	-2.616
Awd	-3.0595

+

Psa	3.8529
vnd	3.6457
ct	3.201
fs(1)h	3.1489
btd	3.1229
zfh2	2.8421
RhoBTB	2.6022
pros	2.5679
CG1124	2.5475
S	2.5424
oc	2.5111
Fur1	2.43
PHDP	2.304
CG31241	2.2802
rux	2.2232
CG14889	2.1752
CG31163	2.1606
HmgZ	2.0795
svp	-2.0404
TER94	-2.1807
corto	-2.3481
olf413	-2.4404
brat	-2.7256
CG31368	-2.7293
mub	-2.9555
Awd	-3.1413
lola	-3.8882

+

lola	3.0121
CG31368	2.8063
Kr-h1	2.7262
svp	2.7055
mub	2.6475
CG5149	2.5248
run	2.4759
tna	2.4302
CG6954	2.4235
CG11153	2.3045
Awd	2.2295
CG6919	2.1324
CG14888	2.067
Psa	-2.0276
rux	-2.093
fs(1)h	-2.141
CG1124	-2.155
Fur1	-2.1588
S	-2.2539
corto	-2.2618
oc	-2.3017
CG14889	-2.4393
zfh2	-2.5884
HmgZ	-3.6328
btd	-3.7627
brat	-3.7716

+

lola	3.3019
CG6919	2.9965
CG31368	2.817
CG5149	2.7675
Kr-h1	2.7647
TER94	2.6286
tna	2.5748
CG11153	2.4795
run	2.3831
CG14888	2.0938
S	-2.0243
rux	-2.0668
oc	-2.3437
corto	-2.5556
fs(1)h	-2.6211
brat	-2.9904
ct	-3.3404
zfh2	-4.4947
CG6954	-4.7244

+

brat	5.2812
ct	4.828
CG31163	4.3345
LpR2	3.6882
vnd	3.6866
zfh2	3.5314
pros	3.4307
Psa	3.3998
fs(1)h	3.3869
CG31241	2.9973
HmgZ	2.9226
Fur1	2.7469
RhoBTB	2.7189
oc	2.6543
Toll-7	2.6161
rux	2.5975
CG14889	2.3054
S	2.2324
CG1124	2.0216
Kr-h1	-2.1439
tna	-2.1793
CG5149	-2.1892
run	-2.2194
Trim9	-2.251
olf413	-2.3821
btd	-3.0293
CG6919	-3.3719

Gene Clustering

- To make the big table we first need to find out all the genes present in at least one of the files
- Make sure not to use factors in `read.delim()`

```
genes <- c()
for(fileNum in 1:5){
  t <- read.delim(paste("2.3_DiffGenes",fileNum,".tsv"
                        ,sep="")
                  ,as.is=TRUE,header=FALSE)
  names(t) <- c("gene","expression")
  genes <- union(genes, t$gene)
}
genes <- sort(genes)
genes
```

Gene Clustering

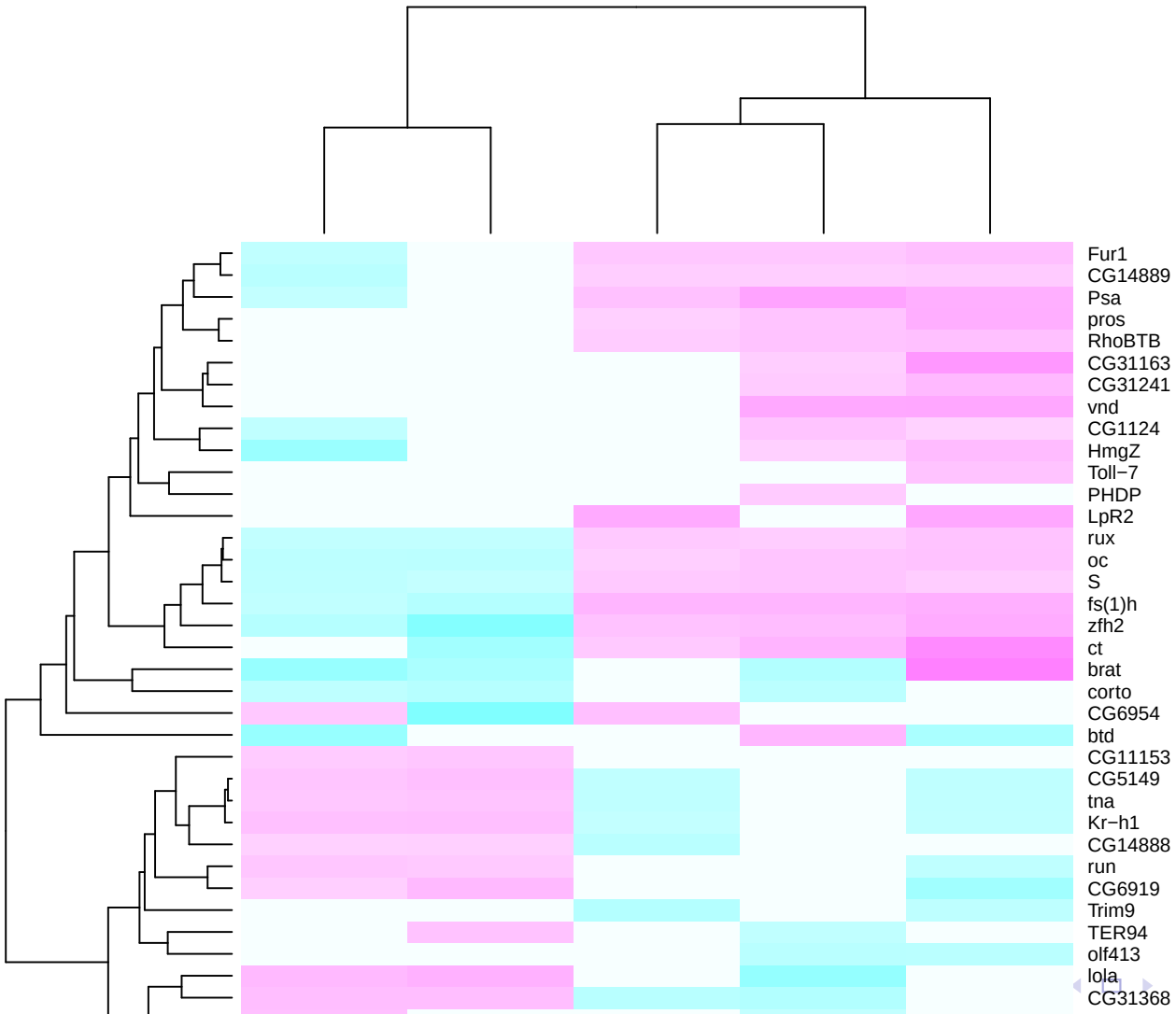
- Using the complete list of genes, we can create the big table and fill in the values:

```
values <- matrix(0, nrow=length(genes),ncol=5)
rownames(values) <- genes
for(fileNum in 1:5){
    t <- read.delim(paste("2.3_DiffGenes",fileNum,".tsv"
                          ,sep=""),as.is=TRUE,header=FALSE)
    names(t) <- c("gene","expression")
    index <- match(t$gene, rownames(values))
    values[index,fileNum] <- t$expression
}
```

Gene Clustering

- Now we can do hierarchical clustering:

```
heatmap(values, scale="none",col=cm.colors(256))
```



Gene Clustering

- In a second part of our analysis, we want to produce the same heatmap but only based on a list of experimentally-verified genes
- The problem is that the data are not formatted in the most convenient way:

genes	citation
oc,run,RhoBTB,CG5149,CG11153,S,Fur1	Segal et al, Development 2001
tna,Kr-h1,rux	Krejci et al, Development 2002

Gene clustering

- We load in this table, and only extract the gene names, then we use them to select a subset of the `values` matrix

```
t.exp <- read.delim("2.3_ExperimentalGenes.tsv",
                    as.is=TRUE)
experim.genes <- unlist(strsplit(t.exp$genes, ","))
```

- `unlist` flattens out a nested list into a single vector
- `strsplit` splits a vector of strings by a custom split character (`,`). The result is a list of split values for each element of the input vector

```
is.experimental <- rownames(values) %in% experim.genes
heatmap(values[is.experimental,],scale="none"
        ,col=cm.colors(256))
```

Gene clustering review

- We load the five tables twice - first to collect gene names, then to load expression values
- Based on the expression table (`values`) we construct a clustered heatmap first on the whole set of genes, then on a selected subset
- Go through the code, try it out and understand it
- Try answering the following questions
  - what is `rownames(values)`
  - why is `rownames(values)[index]` and `t$gene` giving the same output?
  - what is the difference between `rownames(values) %in% experim.genes` and `experim.genes %in% rownames(values)`

Graphics

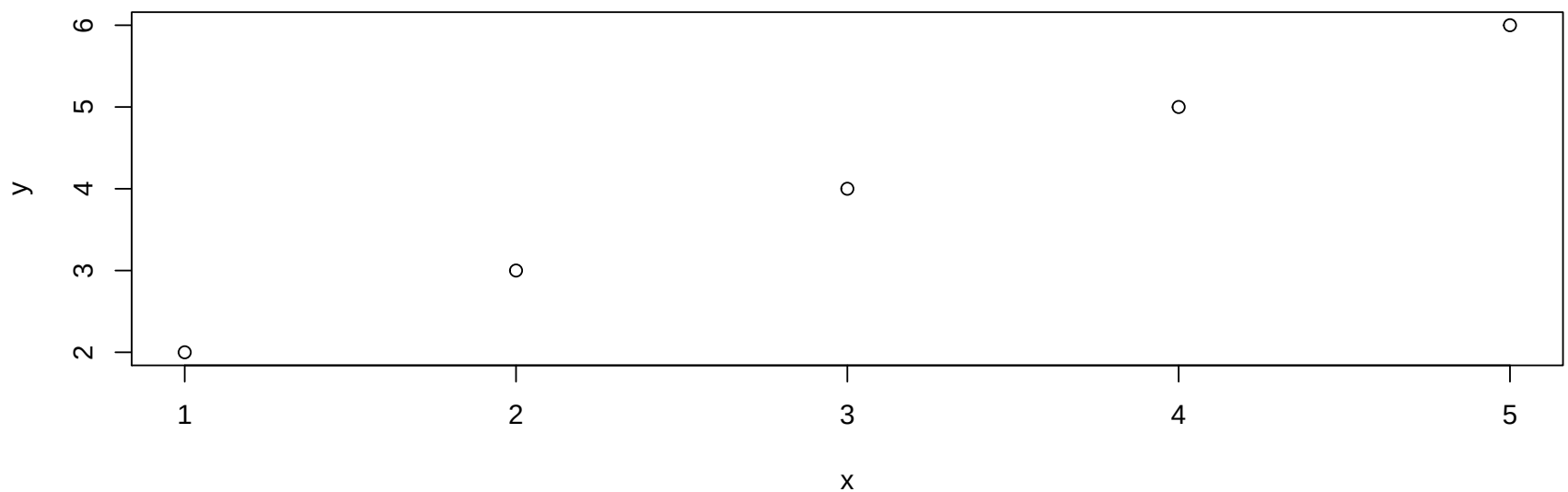
Starting out with R grapics

- R provides several mechanisms for producing graphical output
  - Functionality depends on the level at which the user seeks interaction with R
- High-level graphics
  - Functions compute an appropriate chart based upon the information provided. Optional arguments may tailor the chart as required
  - Interaction is at traditional graphics system level. The user isn't required to know much about anything
- Low-level grpahics
  - The user interacts with the drawing device to build up a picture of the chart piece-by-piece
  - This fine granular control is only required if you seek to do something exceptional
- R graphics produces plots using a painters' model
  - Elements of the graph are added to the canvas one layer at a time, and the picture built up in levels. Lower levels are obscured by higher levels, allowing for blending, masking and overlaying of objects

Essential plotting - plot

- `plot()` is the main function for plotting, it takes x, y values to plot and also lots of graphical parameters (see `?par` for all of them)
- Default plotting

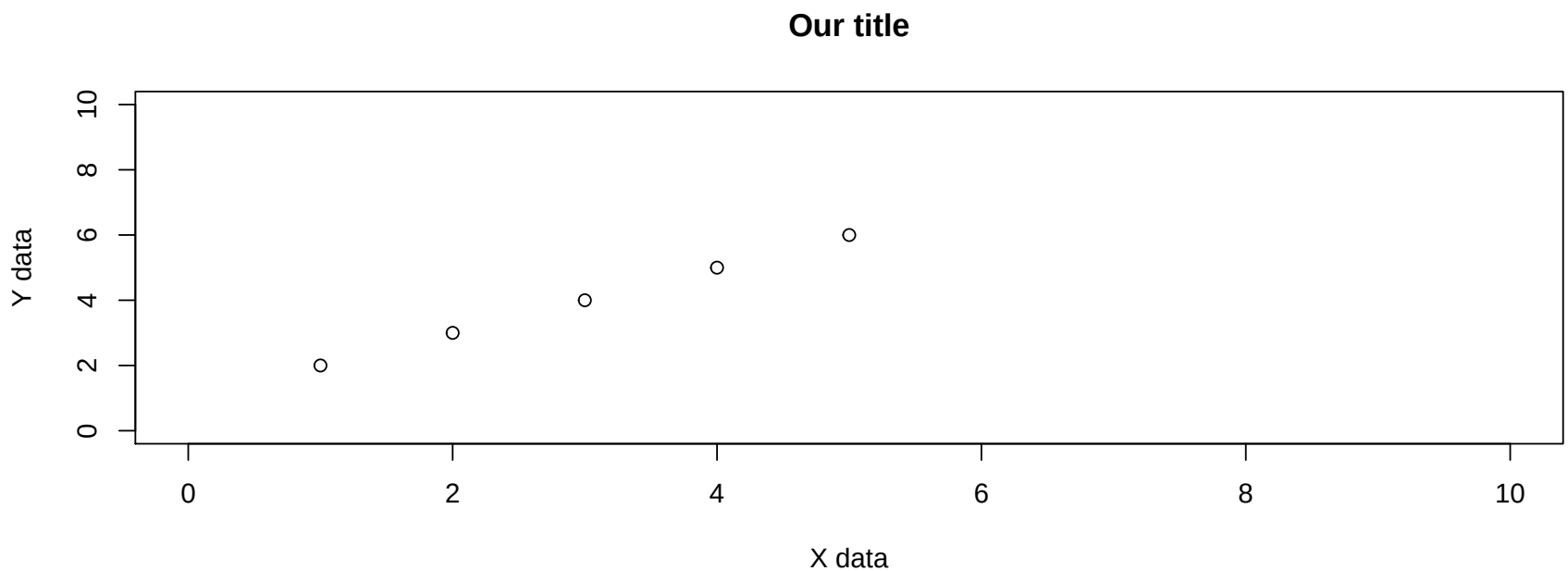
```
x <- 1:5
y <- 2:6
plot(x,y)
```



Essential plotting - plot

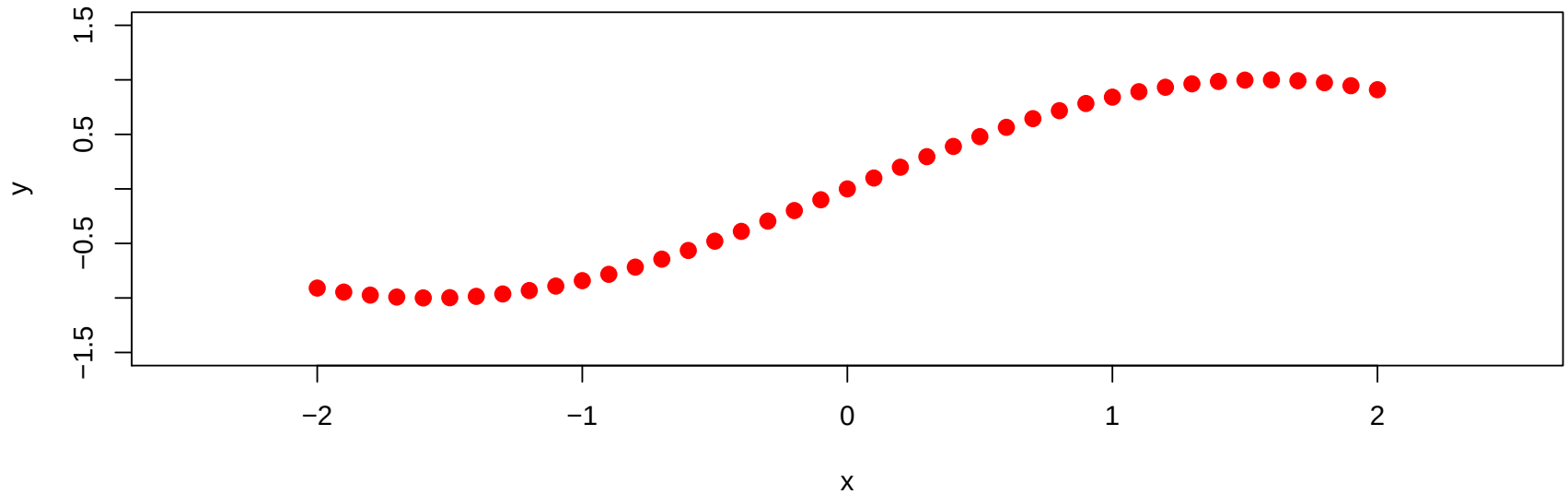
- Custom plotting

```
x <- 1:5
y <- 2:6
plot(x,y, xlab="X data", ylab="Y data",
     xlim=c(0,10),ylim=c(0,10),
     main="Our title")
```

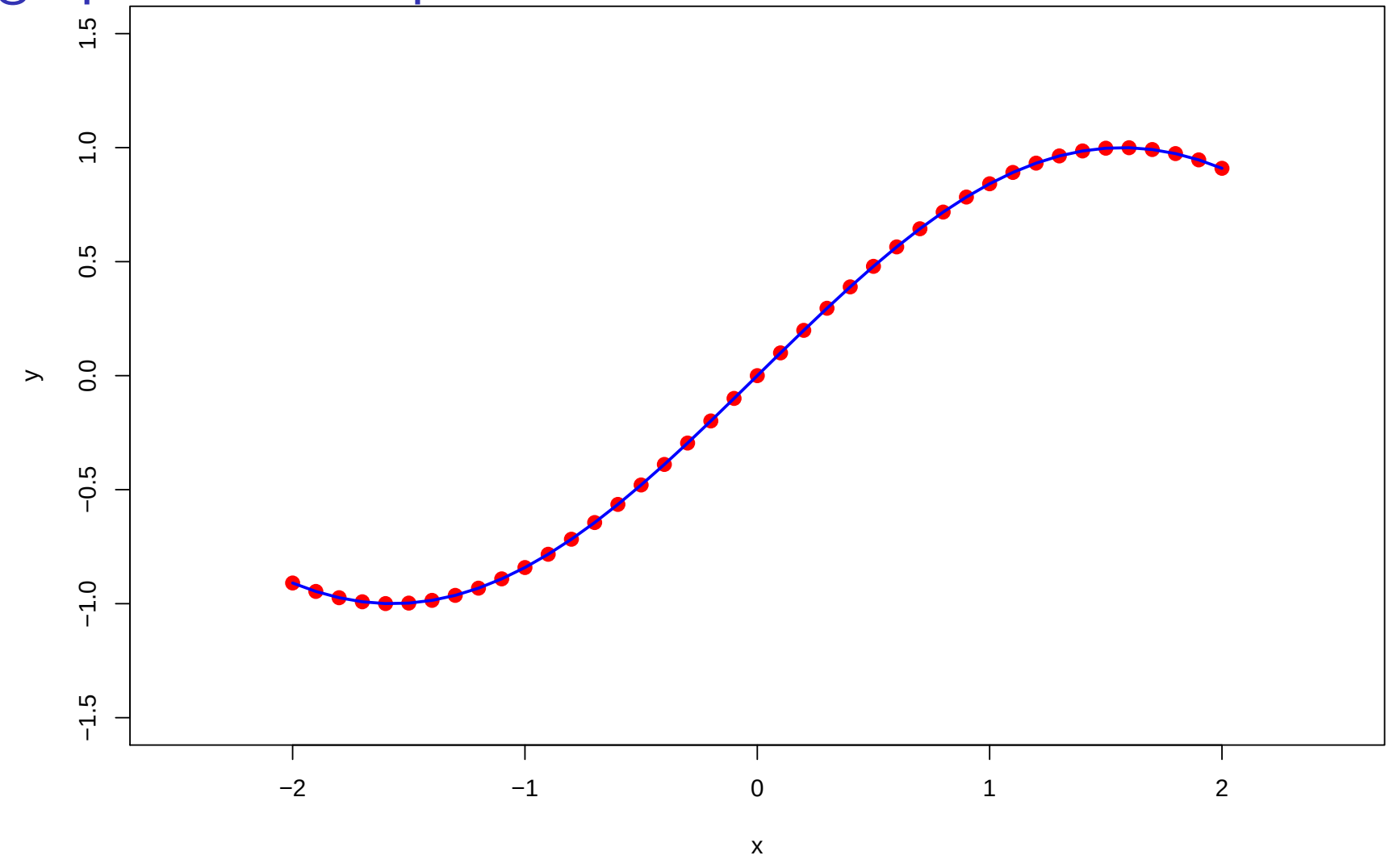


R graphics uses a painters' model

```
x <- seq(-2,2,0.1)
y <- sin(x)
plot(x,y, ylim=c(-1.5,1.5),
     xlim=c(-2.5,2.5),
     col="red",pch=16,
     cex=1.4)
```

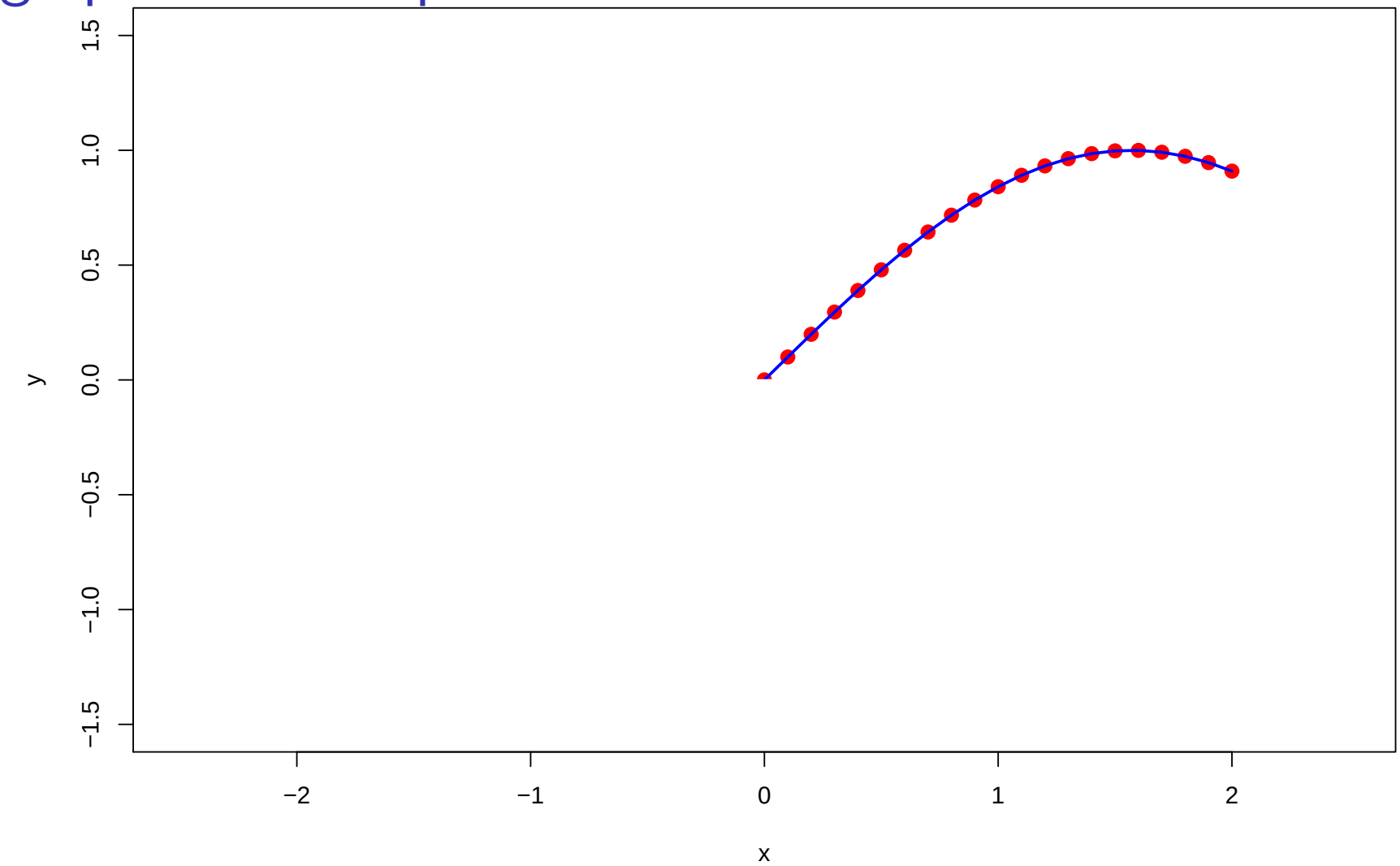


R graphics uses a painters' model




```
lines(x,y,ylim=c(-1.5,1.5),
      xlim=c(-2.5,2.5),col="blue",
      lty=1,lwd=2)
```

R graphics uses a painters' model




```
rect(-2.5,0,2.5,-1.5,
     col="white",border="white")
```

Plotting arguments we used

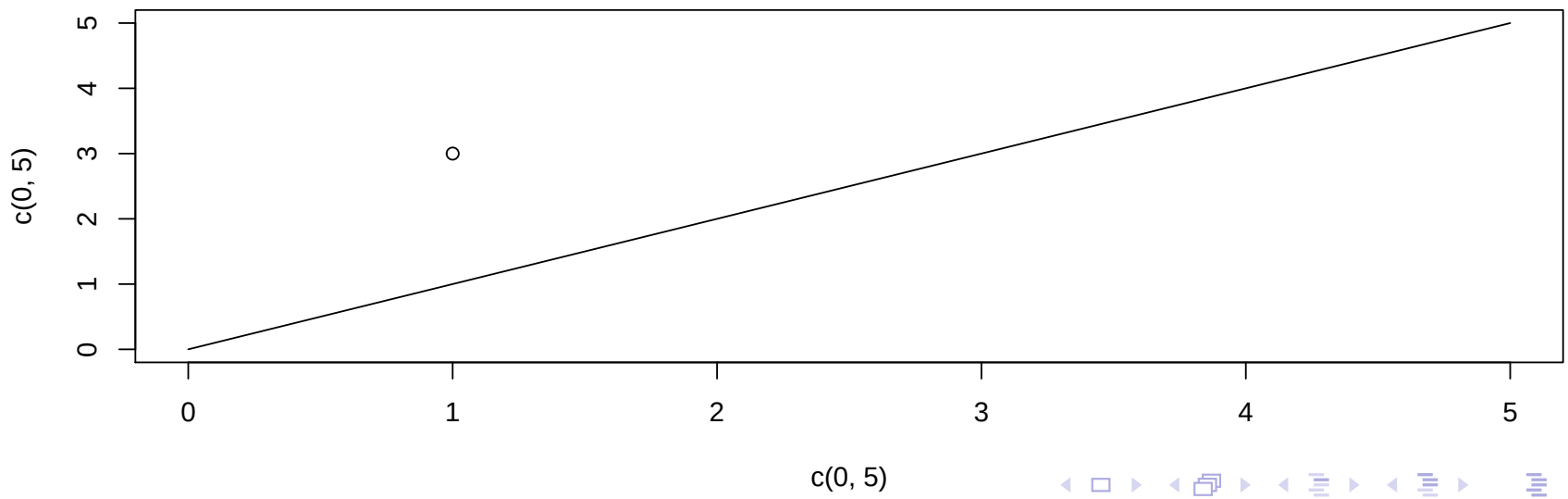
- `xlim`,`ylim`; axis limits
- `col`; line colour
- `pch`; plotting character
- `cex`; character expansion
- `lty`; line type
- `lwd`; line width
- See more arguments with

```
?par
```

Plotting x, y data - plot(), points(), lines()

- `plot` is used to start a new plot, accepts x,y data, but also data from some objects (like linear regression), Use the parameter `type` to draw points, lines etc (see `?plot`)
- `points()` is used to add points to an existing plot
- `lines()` is used to add lines to an existing plot

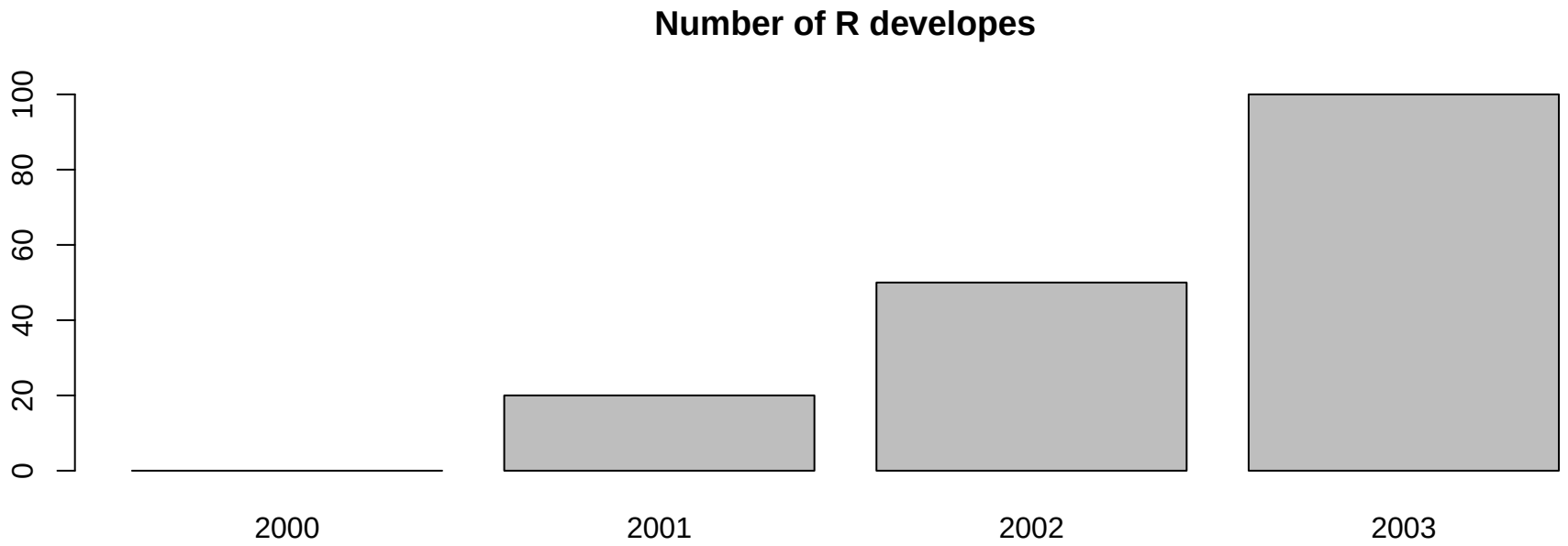
```
# draw as line from (0,0) to (5,5)
plot(c(0,5), c(0,5), type="l")
# add a point at 1,3
points(1,3)
```



Making bar plots barplot()

- Visualizing a vector of data can be done with bar plots, using the function `barplot()`

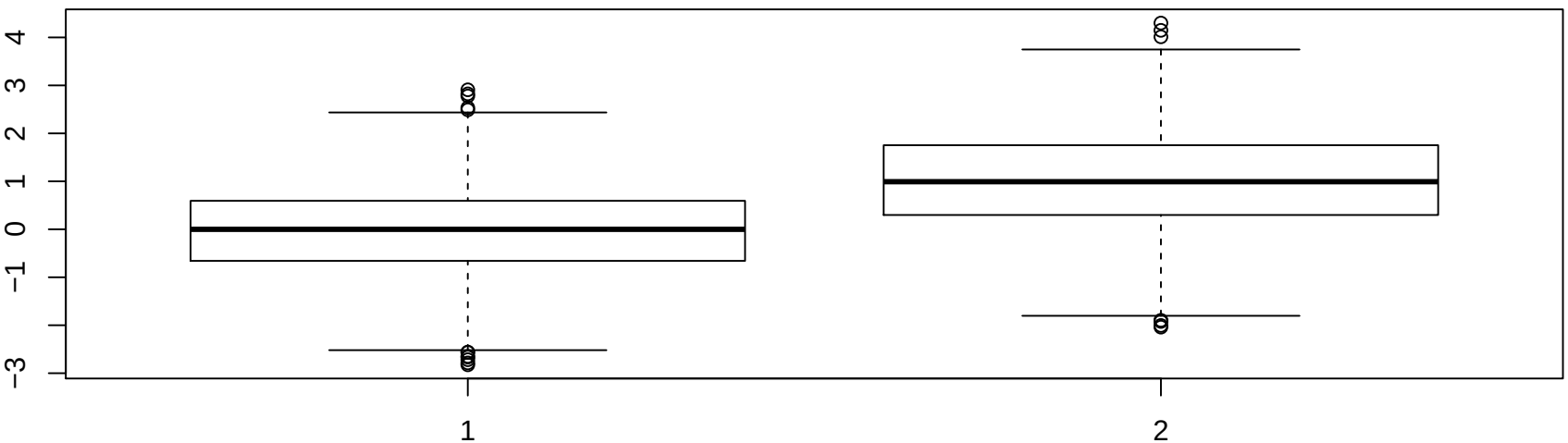
```
data <- c("2000"=0, "2001"=20, "2002"=50,"2003"=100)
barplot(data, main="Number of R developes")
```



Making box plots - boxplot()

- When a spread of data needs to be visualised, we can use boxplots with the function `boxplot()`

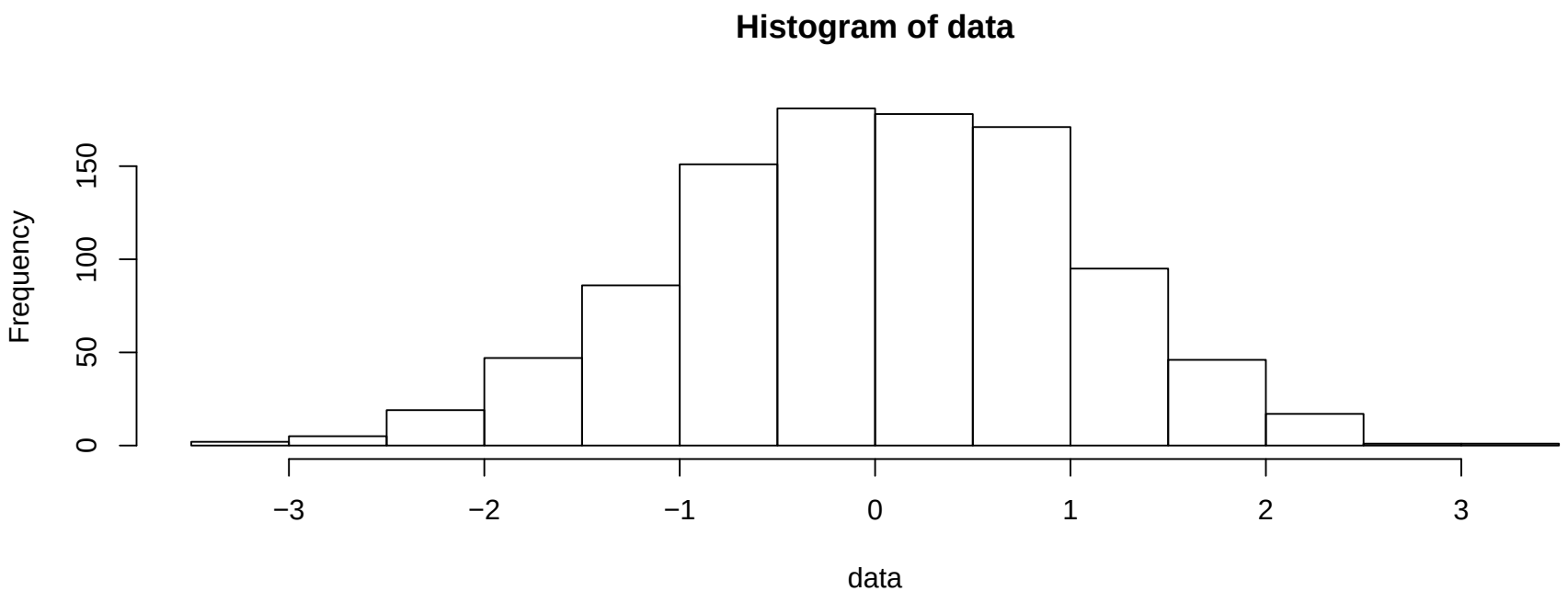
```
data1 <- rnorm(1000,mean=0)
data2 <- rnorm(1000, mean=1)
boxplot(data1,data2)
```



Making histograms - hist()

- When we need to look at the distribution of data, we can visualise it using histograms with the function `hist()`

```
data <- rnorm(1000)
hist(data)
```



Typical plotting workflow

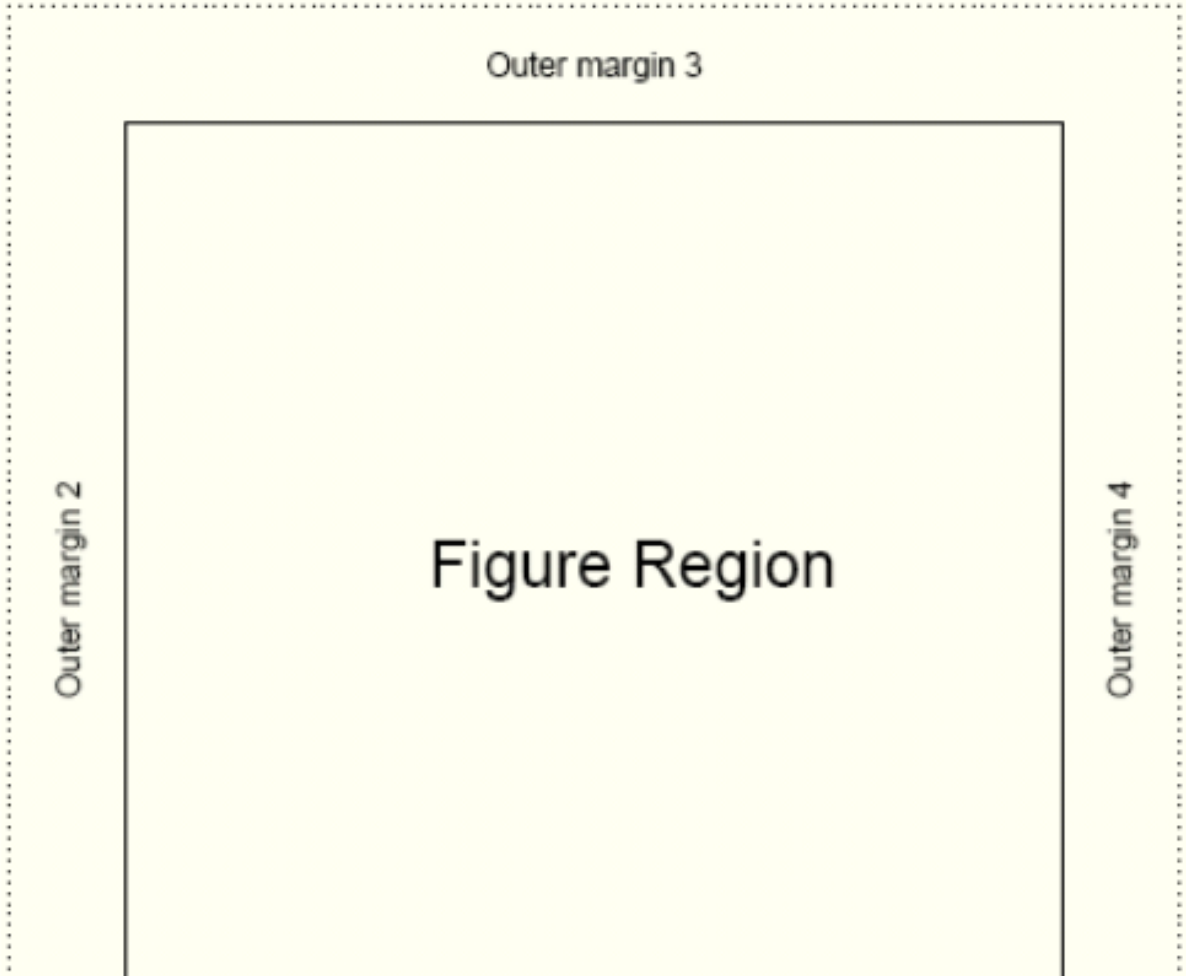
- Set the plot layout and stlye - `par()`
  - set the number of plots you want per page
  - set the outer margins of the figure region (the distance between the edges of the page and the figure region, or between adjacent plots if there are multiple figures per page)
  - set the inner margins of the plot (the distance between the plot axes and the labels and titles)
  - set the styles for the plot (colours, fonts, line styles and weights)
- Draw the plot `plot(x,y,...)`

Setting graphics layout and style - par()

- `par()` top-level graphics function
  - parameter specifies various page settings. These are inherited by subordinate functions, if no other styles are set
  - specific colours and styles may be set globally with `par`, but change ad-hoc in plotting commands
  - the global setting will remain unchanged, and reused in future plotting calls
  - `par` sets the size of page and figure margins (margin spacing is in lines)
  - `par` is responsible for controlling the number of figures that are plotted on a page
  - `par` may set global colouring of axes, text, background, foreground, line styles (solid/dashed), if figures should be boxed or open etc, etc. . .
- type `par()` to get a list of top-down settings that may be set globally

Page settings with par

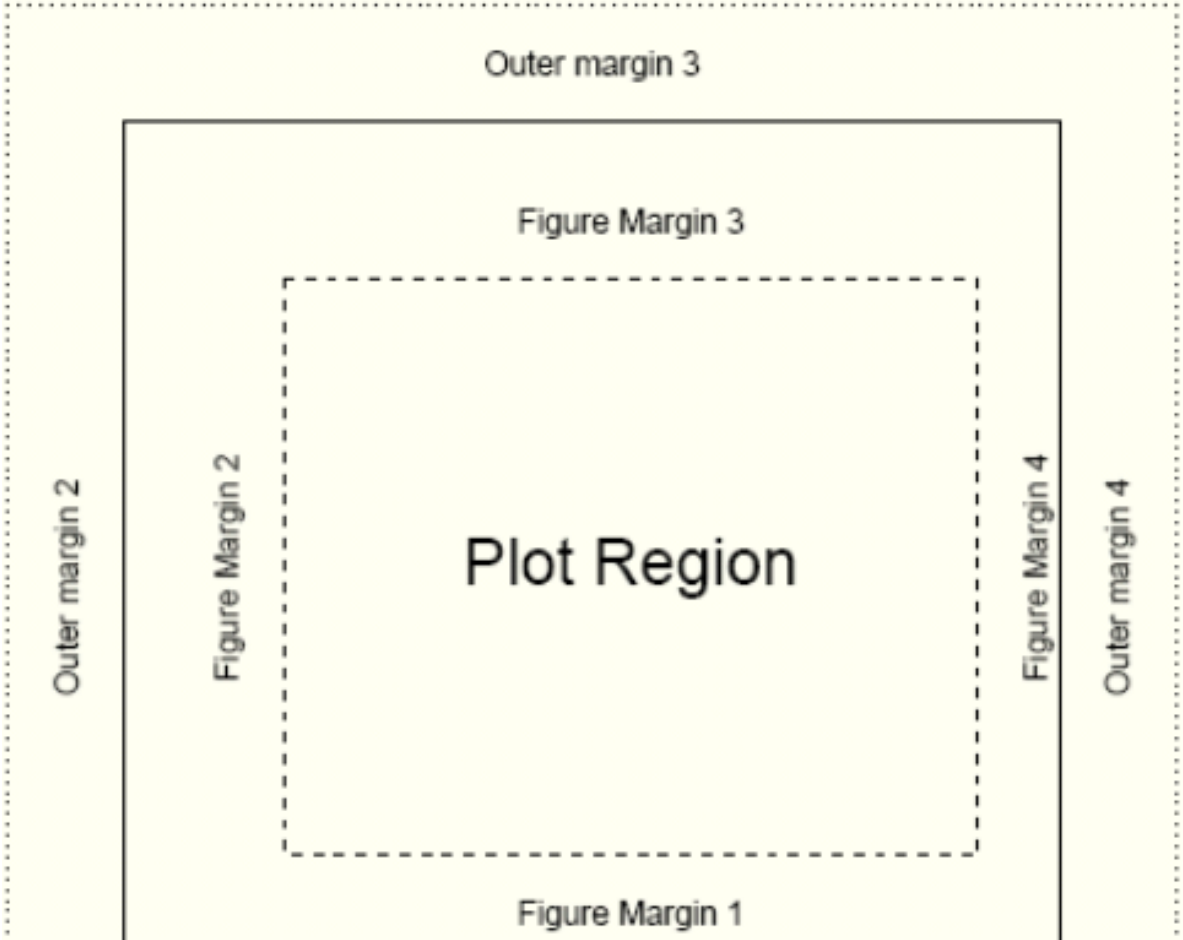
```
#one figure on page
par(mfrow=c(1,1))
#equal outer margins
par(oma=c(2,2,2,2))
```



Page settings with par

```
par(mar=c(5,4,4,2))
```

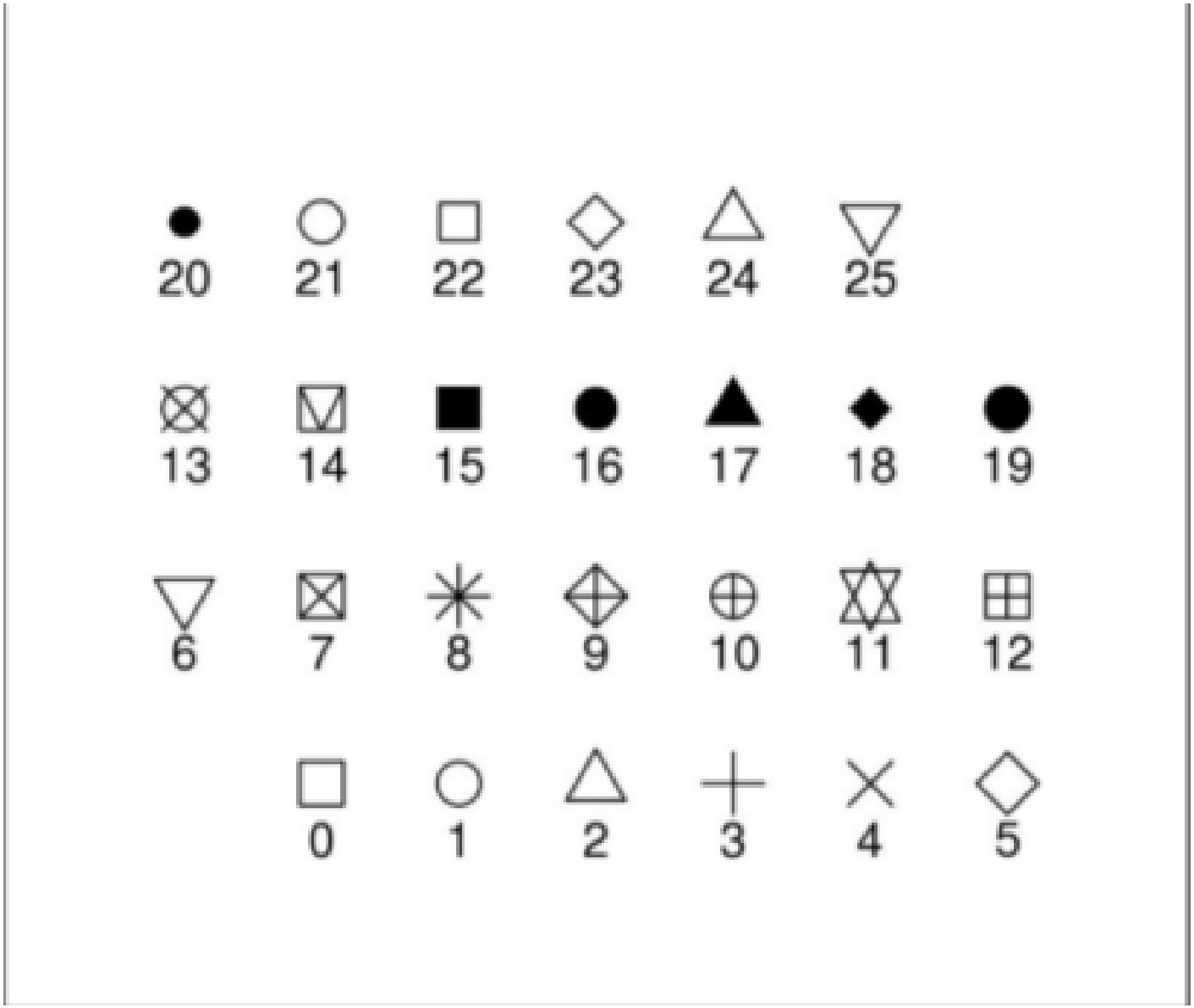
- Sets space for x and y labels, a main title, and a thin margin on the right
  - Numbering: bottom, left, top, right



Plotting characters for plot(), size and orientation

- `pch=`
  - sets on of the 26 standard plotting characters used. Can also use characters, such as `"."`
- `cex=`
  - character expansion. Sets the scaling factor of the printing character
- `las=`
  - Axes label style: 1 = normal, 2= rotated 90 degress.

Plotting characters for plot(), size and orientation



Annotating the plot

- `plot` accepts main title, subtitle, X label, Y label as standard arguments
  - `plot(x,y, main="...", sub="...",xlab="..",ylab="...")`
  - `mtext(text = "...", side="")` allows text to be written directly into the margin of a plot
  - `text(x,y,labels="...")` allows text to be written in the plot at coordinates x,y
  - `legend(x,y,legend=)` produces a legend for the plot

Use of colour in R

- Colour is usually expressed as a hexadecimal code of Red, Green and Blue counterparts
  - no good for humans
- R supports numerous colour palettes which are available through several colour functions
  - `colours()` get inbuilt names of known colours
  - `rgb()` converts red, green, blue intensities to colours

```
rgb(1,1,1)
par(mfrow=c(2,2))
plot(1:10, col="#FF00FF")
plot(1:10, col=rgb(1,0,1))
plot(1:10, col="magenta")
```

Colour ramps and palettes

- Heatmaps use colour depth to convey data values. Cold colours are typically low values, and light colours are high-state values. This is a colour ramp
- R supports numerous graded colour charts. Specify `n` to set the number of gradations required in the palette

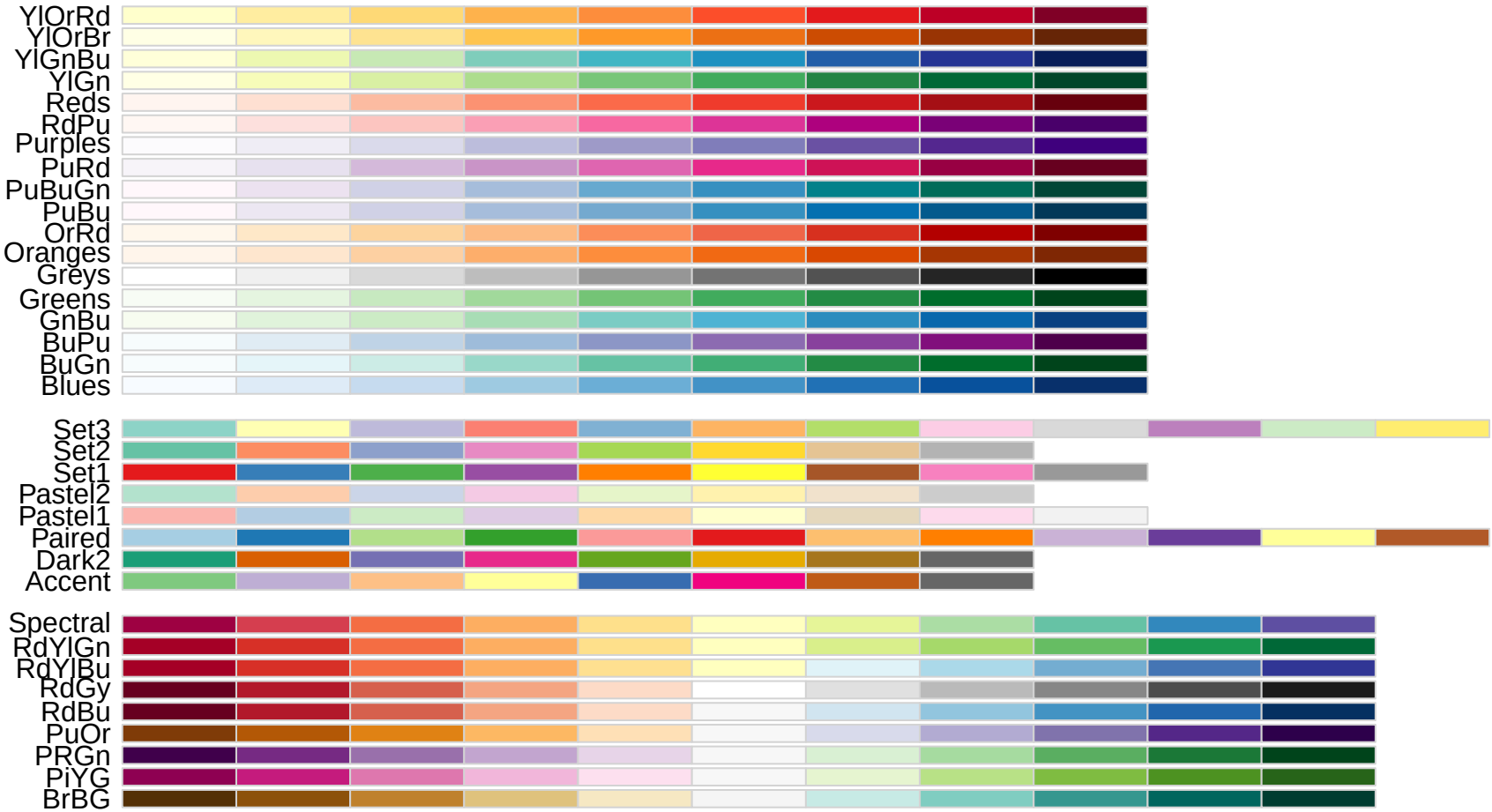
```
rainbow(n)
heat.colors(n)
terrain.colors(n)
topo.colors(n)
cm.colors(n)
```

Colour packages: RColorBrewer

- This add-on package provides a series of well-defined colour palettes. The colours in these palettes are selected to permit maximum visual discrimination
- To access the `RColorBrewer` packages

```
library(RColorBrewer)
display.brewer.all()
#see all available palettes
myCol <- brewer.pal(n,"..")
#n = number of colours, ".." is the palette name
```

Colour packages: RColorBrewer



Saving plots to files

- Unless specified, R plots all graphics to the screen
- To send plots to a file, you need to set up an appropriate graphics device

```
postscript(file = "a_name.ps",...)
pdf(file = "a_name.pdf",...)
jpeg(file = "a_name.jpg",...)
png(file = "a_name.png",...)
```

Saving plots to files

- Each graphics device will have a specific set of arguments to dictate characteristics of the outputted file
  - `height=, width=, horizontal=, res=, paper=`
  - Top tip: A4 @ 300 dpi, portrait, size in pixels
  - Postscript and pdf work in inches by default, A4 = 8.3" x 11.7"
- Graphics devices need closing when printing is finished: `dev.off()`

```
png("tenPoints.png",width=300,height=300)
plot(1:10)
dev.off()
```

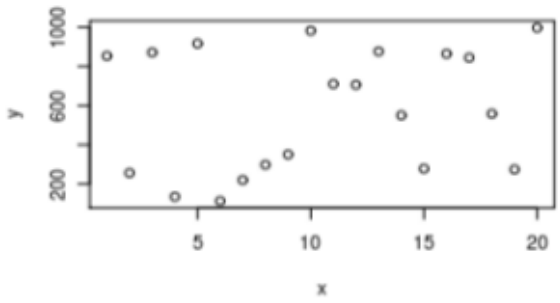
Thoughts when plotting to a file

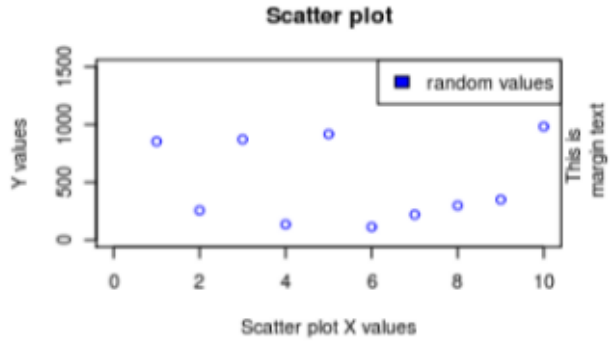
- Its very tempting to send all graphical output to a pdf file. Caution
  - For high resolution publication-quality images you need postscript. Set up postscript file capture with `postscript("a_file.ps", paper="a4")`
  - Postscript images can be converted to JPEG using ghostscript (free to download) for low resolution lab book photos and talks
  - PDF images will grow too large for Acrobat to render if plots contain many data points (e.g. Affymetrix MA plots)
  - Automatically send multiple page outputs to separate image files using `file=somename%02d.jpg`
  - Don't forget to close graphics devices (i.e. the file) by using `dev.off`

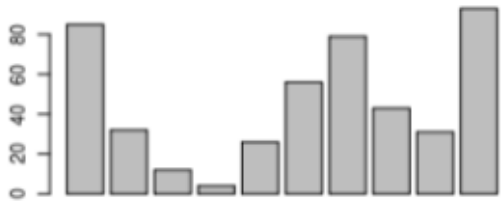
Graphics exercise

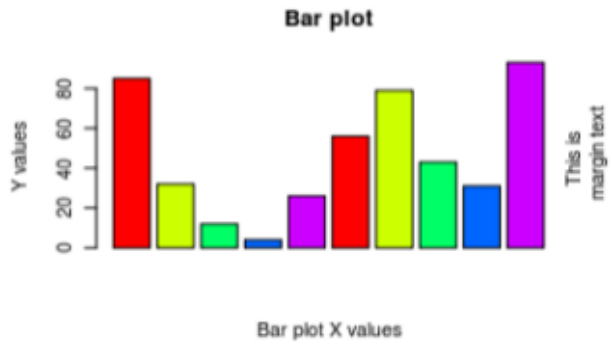
- Exercise
  - Make a full A4 page figure comprising of 6 plots: 2 each of XY plot (`plot()`), barchart (`barplot()`) and box plots (`boxplot()`)
  - The two versions of each plot should consist of: the default plot and a customised plot (change for instance colours, range, captions)
  - Output the completed 6-panel figure to: screen, jpeg, postscript and pdf file
- Suggested route to solution
  - Generate some plotting data appropriate to each type of plot
  - Write the code to produce the six plots, once plotting the data by using the default plotting, one with some customisations
  - To output the plot to screen, jpeg, postscript and pdf you will need to redo the plot multiple times. Create a function to do the plotting and call it by redirecting graphical output to screen, jpeg file, postscript file and pdf file
- An example solution `20_6PanelPlotscript.R`

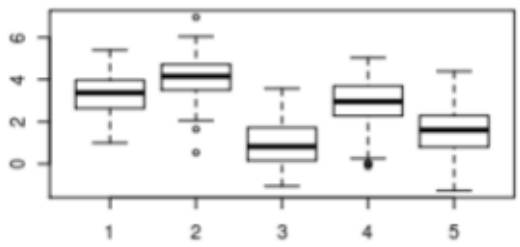
Example output

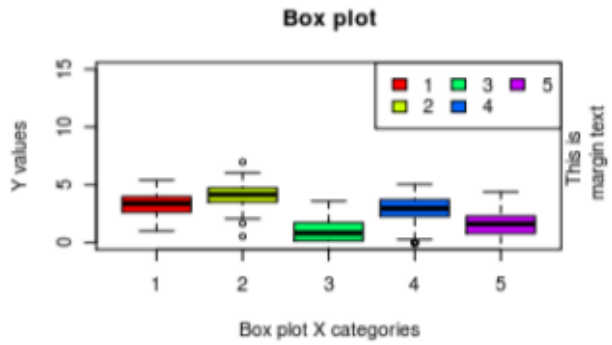












References

References

- Official documentation on:
  - `http://cran.r-project.org/manuals.html`
- A good repository of R recipes
  - Quick-R: `http://www.statmethods.net/`
- Don't forget that many packages come with tutorials (vignettes)
- R forums
  - `http://stackoverflow.com/questions/tagged/r`
  - `http://news.gmane.org/gmane.comp.lang.r.general`
- Plenty of textbooks to choose from, comprehensive list + reviews
  - `http://www.r-project.org/doc/bib/R-books.html`

End of Course

Thanks for your attention

- Please fill-in your feedback so we can improve the course
- The key to learning R is practice, practice, practice!
  - If you don't have your own data yet, look online
  - `http://vincentarelbundock.github.io/Rdatasets/datasets.html`
- Meet with fellow R users
  - `http://www.meetup.com/Cambridge-R-Users-Group-Meetup/`
  - informal, quarterly talks
- Look our for an 'Intermediate R' course