Jung Lab

R for Scientists

Sebastian Heucke

8/27/2019

Contents

Basic concepts in R

The Working Directory (wd)

Like many programs R has a concept of a **working directory** (wd). It is the place where R will look for files to execute and where it will save files, by default. Instead of the setwd() command alternativly in RStudi use the mouse and browse to the directory location *Session –> Set Working Directory –> Choose Directory…*

Command Line Calculation

The command line can be used as a calculator

```
2 + 2
```

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

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

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

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

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

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

Variables

A **variable** is a letter okr word which takes (or contains) a value. We use the assignment 'operartor', <- (shortcut Alt -)

```
x <- 10
x
```

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

```
mynumber <- 25
mynumber
```

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

We can perform arithmetic on variables.

```
sqrt(mynumber)
```

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

We can add variables together.

```
x + mynumber
```

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

We can change the value of an existing variable.

```r
x <- 21
x
```

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

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

```r
x <- mynumber
x
```

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

We can modify the contents of a variable.

```r
mynumber <- mynumber + sqrt(16)
```

Functions

Functions in R perform operations on arguments, the inputs to the function. We have already used sin(x) which returns the sine of x. In this case the function has one argument (x). Arguments are always contained in parentheses, curved brackets (), separated by commas.

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

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

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

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

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

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

Arguments can be named or unnamed, but if they are unnamed they must be ordered.

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

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

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

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

Vectors

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

```r
x <- c("a", "b", "c", "d")
x
```

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

As mentioned, the square brackets [] indicate position within the vector,the **index**. We can extract individual elements by using the [] notation.

```r
x[1]
```

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

```
x[4]
```

```
## [1] "d"
```

We can even put a vector inside the square brackets,vector indexing.

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

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

There are a number of shortcuts to create a vector of numbers.

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

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

```
x <- 3:10
x
```

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

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

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

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

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

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

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

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

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

We have seen some ways of extracting elemets of a vector. We can use these shortcuts to make things easier.

```
letters[3:7]
```

```
## [1] "c" "d" "e" "f" "g"
```

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

```
## [1] "b" "d" "f"
```

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

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

We can add an element to a vector.

```
x <- 3:12
y <- c(x, 1)
y
```

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

We can join vectors together.

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

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

We can remove elements from a vector.

```
letters[-3]
```

```
##  [1] "a" "b" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
## [18] "s" "t" "u" "v" "w" "x" "y" "z"
```

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

```
##  [1] "a" "b" "c" "d" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
## [18] "u" "v" "w" "x" "y" "z"
```

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

```
##  [1] "a" "c" "e" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
## [18] "u" "v" "w" "x" "y" "z"
```

We can modify the contents of a vector.

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

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

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

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

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

Vector Arithmetic

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

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

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

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

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

Adding two vectors.

```
y+z
```

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

Vectors and Naming

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

```r
gene_names <- c("Pax6", "Beta-actin", "FoxP2", "Hox9")
```

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

```r
gene_expression <- c(0, 3.2, 1.2, -2)
gene_expression
```

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

```r
names(gene_expression) <- gene_names
gene_expression
```

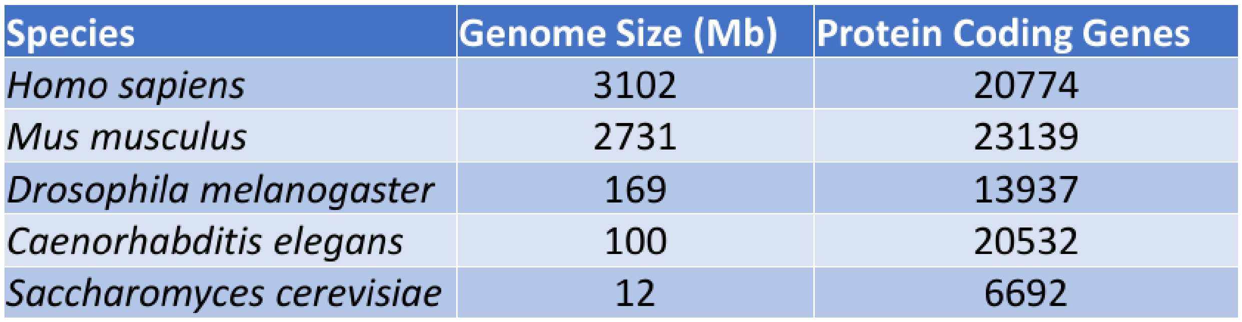
```
##       Pax6 Beta-actin      FoxP2       Hox9
##        0.0        3.2        1.2       -2.0
```

We can also use the **names** function to get a vector of the names of an object.

```r
names(gene_expression)
```

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

Lets translate the following table into R vectors and do some calculations with them.



Species	Genome Size (Mb)	Protein Coding Genes
Homo sapiens	3102	20774
Mus musculus	2731	23139
Drosophila melanogaster	169	13937
Caenorhabditis elegans	100	20532
Saccharomyces cerevisiae	12	6692

```r
genome_size <- c(3102, 2731, 169, 100, 12)
coding_genes <- c(20774, 23139, 13937, 20532, 6692)
species_name <- c("Homo sapiens", "Mus musculus", "Drosophila melanogaster",
                  "Caenorhabditis elegans", "Saccharomyces cerevisiae")
names(genome_size) <- species_name
names(coding_genes) <- species_name
```

```r
genome_table <- data.frame(genome_size, coding_genes)
genome_table
```

```
##                          genome_size coding_genes
## Homo sapiens                    3102        20774
## Mus musculus                    2731        23139
## Drosophila melanogaster          169        13937
## Caenorhabditis elegans           100        20532
## Saccharomyces cerevisiae          12         6692
```

With the assumption that a protein coding gene has an average length of 1.5 kilobases, to calculate the number of coding bases, we need to use the same scale as we use for genome size: 1.5 kilobases is 0.0015

Megabases.

```
coding_bases <- coding_genes * 1500
coding_bases
```

```
##             Homo sapiens             Mus musculus  Drosophila melanogaster
##                 31161000                 34708500                 20905500
##   Caenorhabditis elegans Saccharomyces cerevisiae
##                 30798000                 10038000
```

To calculate the percentage of coding bases in each genome:

```
coding_pc <- (coding_bases / (genome_size *1000000)) * 100
coding_pc
```

```
##             Homo sapiens             Mus musculus  Drosophila melanogaster
##                 1.004545                 1.270908                12.370118
##   Caenorhabditis elegans Saccharomyces cerevisiae
##                30.798000                83.650000
```

To compare human to yeast:

```
coding_bases[1] / coding_bases[5]
```

```
## Homo sapiens
##     3.104304
```

```
genome_size[1] / genome_size[5]
```

```
## Homo sapiens
##        258.5
```

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

```
names(coding_pc) <- NULL
coding_pc
```

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

Explicit Coercion

Objects can be explicitly coerced from one class to another using the as.* functions.

```
x <- 6
class(x)
```

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

```
as.numeric(x)
```

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

```
as.logical(x)
```

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

```
as.character(x)
```

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

Sometimes, R can't figure out how to coerce an object and this can result in *NA's* being produced.

```
x <- c("a", "b", "c")
as.numeric
```

```
## function (x, ...)  .Primitive("as.double")
```

When nonsensical coersion takes place, you will usually get a warning from R.

R tricks and tips

Getting Help

To get help on any R function, type **?** followed by the function name.

```
?seq
```

This retrieves the syntax and arguments for the function. You can see the default order of arguments here. The help page also tells you which **package** it belongs to.

There will typically be example usage, which you can test using the **example** function.

```
example(seq)
```

```
##
## seq> seq(0, 1, length.out = 11)
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
##
## seq> seq(stats::rnorm(20)) # effectively 'along'
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## seq> seq(1, 9, by = 2)     # matches 'end'
## [1] 1 3 5 7 9
##
## seq> seq(1, 9, by = pi)    # stays below 'end'
## [1] 1.000000 4.141593 7.283185
##
## seq> seq(1, 6, by = 3)
## [1] 1 4
##
## seq> seq(1.575, 5.125, by = 0.05)
##  [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.925 1.975 2.025 2.075
## [12] 2.125 2.175 2.225 2.275 2.325 2.375 2.425 2.475 2.525 2.575 2.625
## [23] 2.675 2.725 2.775 2.825 2.875 2.925 2.975 3.025 3.075 3.125 3.175
## [34] 3.225 3.275 3.325 3.375 3.425 3.475 3.525 3.575 3.625 3.675 3.725
## [45] 3.775 3.825 3.875 3.925 3.975 4.025 4.075 4.125 4.175 4.225 4.275
## [56] 4.325 4.375 4.425 4.475 4.525 4.575 4.625 4.675 4.725 4.775 4.825
## [67] 4.875 4.925 4.975 5.025 5.075 5.125
##
## seq> seq(17) # same as 1:17, or even better seq_len(17)
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
```

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

```
??eatmap
```

Interaction with RStudio

A **;** at the end of line enables multiple commands to be placed on one line of text. The # symbol indicates text as a comment and not executed. The + is used as command line wrap, R will wait for you to complete an expression. Use **esc** to clear input line and try again. **Ctrl-L** to clear window. Use the **TAB** key for command auto completion. Use **up and down arrows** to scroll through the command history.

R Packages

R comes ready with various libraries of functions called **packages**. The **stats** package for instance includes the sd() function, which calculates the standard deviation of a vector. There are 1000s of additional packages provided by third parties, and the packages cam be found in numerous server locations on the web called repositories. The two repositories you will come across the most are **CRAN** the Comprehensive R Archive Network and **Bioconductor**.

Install Packages

Use **install.packages(..) function. Use** library(..) function to load the newly installed features. Use **library()** to list the packages you've got installed locally.

Data Structures

Character, Numeric and Logical Data Types

Although the basic unit of R is a vector, we usually handle data in **data frames**. A data frame is a set of obervations of a set of variables, in other words, the outcome of an experiment. For example, we might want to analyse information about a set of patiens. To start with, let's say we have ten patients and for each one we know their name, sex, age, weight and whether they give consenst for their data to be made public. We are going to create a data frame called 'patients', which wil have ten rows (observations) and seven columns (variables). The columns must all be equal lengths.

```
age <- c(50, 21, 35, 45, 28, 31, 42, 33, 57, 62)
weight <- c(70.8, 67.9, 75.3, 61.9, 72.4, 69.9, 63.5, 71.5, 73.2, 64.8)
firstName <- c("Adam", "Eve", "John", "Mary", "Peter", "Paul", "Joanna",
"Matthew", "David", "Sally")
secondName <- c("Jones", "Parker", "Evans", "Davis", "Baker", "Daniels",
"Edwards", "Smith", "Roberts", "Wilson")
 consent <- c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, TRUE)
```

Vectors can only contain one type of data, we cannot mix numbers, characters and logical values in the same vector. If we try this, R will convert everything to characters. We can see the type of a particular vector using the **mode** function.

```
mode(firstName)
```

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

```
mode(age)
```

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

```
mode(consent)
```

```
## [1] "logical"
```

Factors

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

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

```
##  [1] "Male"   "Female" "Male"   "Female" "Male"   "Male"   "Female"
##  [8] "Male"   "Male"   "Female"
```

```
factor(sex)
```

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

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

Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
dim(m)
```

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

```
attributes(m)
```

```
## $dim
## [1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

Matrices can also be created directly from vectors by adding a dimension attribute.

```
m <- 1:10
dim(m) <- c(2, 5)
m
```

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

Matrices can be created by *column-binding* or *row-binding* with the *cbind*() and *rbind()* functions.

```
x <- 1:3
y <- 10:12
cbind(x,y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Lists

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

```
a <- 1:10
b  <- matrix(runif(100), ncol=10, nrow = 10)
d <- data.frame(a, month.name[1:10])

myList <- list(ls.obj.1=a, ls.obj.2=b, ls.obj.3=d)
summary(myList)
```

```
##          Length Class      Mode
## ls.obj.1  10    -none-     numeric
## ls.obj.2 100    -none-     numeric
## ls.obj.3   2    data.frame list
```

```
names(myList)
```

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

We can use the dollar syntax to access list items.

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

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

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

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

The second double approach with double [[]] is necessary because lists are in fact like vectors, they can only contain one type of object. But one of the types the can contain is a list. So any lilst like the above is acutally a list of lists; the first element myList[1] is a list containing a vector, and so we need double indexing to actually get the vector.

Missing Values

Missing values are denoted by NA or NaN for q undefined mathematical operations. * is.na() is used to test objects if they are NA * is.nan() is used to test for NaN * NA values have a class also, so there are integer NA, character NA, etc. * A NaA value is also NA but the converse is not true

```
# create a vector with NAs in it
x <- c(1, 2, NA, 10, 3)
# return a logical vector indicating which elements are NA
is.na(x)
```

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

```r
# return a logical vector indicating which elements are NaN
is.nan(x)
```

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

```r
# now create a vector with both NA and NaN values
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

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

```r
is.nan(x)
```

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

Data Frame

Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Data frames 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. Matrices must have every element be the same class. In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called row.names which indicate information about each row of the data frame. Data frames are usually created by reading in a dataset using the *read.table()* or *read.csv()*. However, data frames can also be created explicitly with the *data.frame()* function or they can be coerced from other types of objects like lists. Data frames can be converted to a matrix by calling *data.matrix()*.

```r
patients <- data.frame(firstName, secondName, paste(firstName, secondName),
                       sex, age, weight, consent)
patients
```

```
##    firstName secondName paste.firstName..secondName.    sex age weight
## 1       Adam      Jones                   Adam Jones   Male  50   70.8
## 2        Eve     Parker                   Eve Parker Female  21   67.9
## 3       John      Evans                   John Evans   Male  35   75.3
## 4       Mary      Davis                   Mary Davis Female  45   61.9
## 5      Peter      Baker                  Peter Baker   Male  28   72.4
## 6       Paul    Daniels                 Paul Daniels   Male  31   69.9
## 7     Joanna    Edwards               Joanna Edwards Female  42   63.5
## 8    Matthew      Smith                Matthew Smith   Male  33   71.5
## 9      David    Roberts                David Roberts   Male  57   73.2
## 10     Sally     Wilson                 Sally Wilson Female  62   64.8
##    consent
## 1     TRUE
## 2     TRUE
## 3    FALSE
## 4     TRUE
## 5    FALSE
## 6    FALSE
## 7    FALSE
## 8     TRUE
## 9    FALSE
## 10    TRUE
```

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

```
patients$age
```

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

R has inferred the names of our data frame variables from the names of vectors or the commands. We can name the variables after we have created a data frame using the **names** function, and we can use the same function to see the names.

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

```
## [1] "First_Name"  "Second_Name" "Full_Name"   "Sex"         "Age"
## [6] "Weight"      "Consent"
```

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

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

```
## [1] "First_Name"  "Second_Name" "Full_Name"   "Sex"         "Age"
## [6] "Weight"      "Consent"
```

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

```
patients$first_Name
```

```
## NULL
```

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

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

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

```
patients$First_Name
```

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

Indexing Data Frames and Matrices

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

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

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

```
patients[1,]
```

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

Advanced Indexing

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

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

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

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

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

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

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

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

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

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

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

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

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

Managing data frames with the *dplyr* package

Data frames

The *data frame* is a key data structure in statistics and in R. The basic structure of a data frame is that there is one observation per row and each column represents a variable, a measure, feature, or characteristic of that observation. Given the importance of managing data frames, it's important that we have good tools for dealing with them. In previous chapters we have already discussed some tools like the *subset()* function and the use of [] and $ operators to extract subsets of data frames. However, other operations, like filtering, re-ordering, and collapsing, can often be tedious operations in R whose syntax is not very intuitive. The *dplyer* package is designed to mitigate a lot of these problems and to provide a highly optimized set of routines specifically for dealing with data frames. One important contribution of the *dplyr* package is that it provides a "grammar" for data manipulation and for operating on data frames. With this grammar, you can sensibly communicate what it is that you are doing to a data frame that other people can understand.

Some of the key "verbs" provided by the dplyr package are:

- select: return a subset of the columns of a data frame, using flexible notation
- filter: extract a subset of rows from a data frame based on logical conditions
- arrange: reorder rows of a data frame
- rename: rename variables in a data frame
- mutate: add new variables/columns or transform existing variables
- summarize: generate summary statistics of different variables in the data frame
- %>% the "pipe" operator is used to connect multiple verb actions together into a pipeline

```
library(dplyr)

chicago <- readRDS("chicago.rds")
dim(chicago)
```

```
## [1] 6940    8
```

```
str(chicago)
```

```
## 'data.frame':    6940 obs. of  8 variables:
##  $ city      : chr  "chic" "chic" "chic" "chic" ...
##  $ tmpd      : num  31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...
##  $ dptp      : num  31.5 29.9 27.4 28.6 28.9 ...
##  $ date      : Date, format: "1987-01-01" "1987-01-02" ...
##  $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
##  $ pm10tmean2: num  34 NA 34.2 47 NA ...
##  $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
##  $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

The *select()* function can be used to select columns of da data frame that you want to focus on. Often you'll have a large data frame containing "all" of the data, but *given* analysis might only use a subset of variables or observations. The *select()* function allows you to get the few columns you might need.

Suppose we wanted to take the first 3 columns only. Theree are a few ways to do this. We could for example use numerical indices. But we can also use the names directly.

```
names(chicago)[1:3]
```

```
## [1] "city" "tmpd" "dptp"
```

```
subset <- select(chicago, city:dptp)
head(subset)
```

```
##   city tmpd   dptp
## 1 chic 31.5 31.500
## 2 chic 33.0 29.875
## 3 chic 33.0 27.375
## 4 chic 29.0 28.625
## 5 chic 32.0 28.875
## 6 chic 40.0 35.125
```

The *filter()* function is used to extract subsets of rows from a data frame. This function is similar to the existing *subset()* function in R but is quite a bit faster. Suppose we wanted to extract the rows of the chicago data frame where the levels of PM2.5 are greater than 30.

```
chic.f <- filter(chicago, pm25tmean2 > 30)
str(chic.f)
```

```
## 'data.frame':    194 obs. of  8 variables:
##  $ city      : chr  "chic" "chic" "chic" "chic" ...
##  $ tmpd      : num  23 28 55 59 57 57 75 61 73 78 ...
##  $ dptp      : num  21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ...
##  $ date      : Date, format: "1998-01-17" "1998-01-23" ...
##  $ pm25tmean2: num  38.1 34 39.4 35.4 33.3 ...
##  $ pm10tmean2: num  32.5 38.7 34 28.5 35 ...
##  $ o3tmean2  : num  3.18 1.75 10.79 14.3 20.66 ...
##  $ no2tmean2 : num  25.3 29.4 25.3 31.4 26.8 ...
```

```
summary(chic.f$pm25tmean2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   30.05   32.12   35.04   36.63   39.53   61.50
```

We can place an arbitrarily complex logical sequence inside of *filter()*, so we could for example extract the rows where PM2.5 is greater then 30 and temperature is greater than 80 degrees Fahrenheit.

```
chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
select(chic.f, date, tmpd, pm25tmean2)
```

```
##          date tmpd pm25tmean2
## 1  1998-08-23   81   39.60000
## 2  1998-09-06   81   31.50000
## 3  2001-07-20   82   32.30000
## 4  2001-08-01   84   43.70000
## 5  2001-08-08   85   38.83750
## 6  2001-08-09   84   38.20000
## 7  2002-06-20   82   33.00000
## 8  2002-06-23   82   42.50000
## 9  2002-07-08   81   33.10000
## 10 2002-07-18   82   38.85000
## 11 2003-06-25   82   33.90000
## 12 2003-07-04   84   32.90000
## 13 2005-06-24   86   31.85714
## 14 2005-06-27   82   51.53750
## 15 2005-06-28   85   31.20000
## 16 2005-07-17   84   32.70000
## 17 2005-08-03   84   37.90000
```

The *arrange()* function is used to reorder rows of a data frame according to one of the variables/columns. Reordering rows of a data frame (while preserving corresponding order of the other columns) is normally a pain to do in R. The *arrange()* function simplifies the process quite a bit.

Here we can order the rows of the data frame by data, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation.

```
chicago <- arrange(chicago, date)
head(select(chicago, date, pm25tmean2), 3)
```

```
##         date pm25tmean2
## 1 1987-01-01         NA
## 2 1987-01-02         NA
## 3 1987-01-03         NA
```

```
tail(select(chicago, date, pm25tmean2), 3)
```

```
##            date pm25tmean2
## 6938 2005-12-29    7.45000
## 6939 2005-12-30   15.05714
## 6940 2005-12-31   15.00000
```

Columns can be arranged in descending order too by useing the special *desc()* operator.

```
chicago <- arrange(chicago, desc(date))
head(select(chicago, date, pm25tmean2), 3)
```

```
##         date pm25tmean2
## 1 2005-12-31   15.00000
## 2 2005-12-30   15.05714
## 3 2005-12-29    7.45000
```

Renaming a variable in a data frame in R is surprisingly hard to do. The *rename()* function is designed to make this process easier.

Here you can see the names of the first five variables in the chicago data frame.

```
head(chicago[, 1:5], 3)
```

```
##   city tmpd dptp       date pm25tmean2
## 1 chic   35 30.1 2005-12-31   15.00000
## 2 chic   36 31.0 2005-12-30   15.05714
## 3 chic   35 29.4 2005-12-29    7.45000
```

The dptp column is supposed to represent the dew point temperature and the pmtmean2 column provides the PM2.5 data. However, these names are pretty obscure or awkward and probably be renamed to something more sensible.

```
chicago <- rename(chicago, dewpoint = dptp, pm25 = pm25tmean2)
head(chicago[, 1:5], 3)
```

```
##   city tmpd dewpoint       date     pm25
## 1 chic   35     30.1 2005-12-31 15.00000
## 2 chic   36     31.0 2005-12-30 15.05714
## 3 chic   35     29.4 2005-12-29  7.45000
```

The syntax inside the *rename()* function is to have the new name on the left-hand side of the = sign and the old name on the right-hand side.

The *mutate()* function exists to compute transformations of variables in a data frame. Often, you want to create new variables that are derived from existing varibles and *mutate()* provides a clean interface for doing that.

For example, with air pollution data, we often want to *detrend* the data by subtracting the mean form the data. That way we can look at whether a given day's air pollution level is higher than or less then average (as opposed to looking at its absolute level).

Here we create a pm25detrend variable that substracts the mean from the pm25 variable.

```
chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm=TRUE))
head(chicago)
```

```
##   city tmpd dewpoint       date     pm25 pm10tmean2  o3tmean2 no2tmean2
## 1 chic   35     30.1 2005-12-31 15.00000       23.5  2.531250  13.25000
## 2 chic   36     31.0 2005-12-30 15.05714       19.2  3.034420  22.80556
## 3 chic   35     29.4 2005-12-29  7.45000       23.5  6.794837  19.97222
## 4 chic   37     34.5 2005-12-28 17.75000       27.5  3.260417  19.28563
## 5 chic   40     33.6 2005-12-27 23.56000       27.0  4.468750  23.50000
## 6 chic   35     29.6 2005-12-26  8.40000        8.5 14.041667  16.81944
##   pm25detrend
## 1   -1.230958
## 2   -1.173815
## 3   -8.780958
## 4    1.519042
## 5    7.329042
## 6   -7.830958
```

There is also the related *transmute()* function, which does the same thing as *mutate()* but then *drops all non-transformed variables.*

Here we detrend the PM10 and ozone(O3) variables.

```
head(transmute(chicago, pm10detrend=pm10tmean2 - mean(pm10tmean2, na.rm = TRUE),
  o3detrend = o3tmean2 - mean(o3tmean2, na.rm = TRUE)))
```

```
##   pm10detrend  o3detrend
## 1  -10.395206 -16.904263
## 2  -14.695206 -16.401093
## 3  -10.395206 -12.640676
## 4   -6.395206 -16.175096
## 5   -6.895206 -14.966763
## 6  -25.395206  -5.393846
```

Note that there are only two columns in the transmuted data frame.

The *group_by()* function is used to generate summmary statistics from the data frame within strata defined by a variable. For example, in this air pollution dataset, you might want to know what the average annual level of PM2.5 is. So the stratum is the year, and that is something we can derive from the date variable. In conjunction with the *group_by()* function we often use the *summarize()* function (or *summarise*() for some parts of the world).

The general operation here is a combination of splitting a data frame into separate pieces defined by variable or group of variables (*group_by()*), and then applying a summary function across those subsets (summarize()).

```
chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900)
years <- group_by(chicago, year)
 summarize(years, pm25 = mean(pm25, na.rm = TRUE),
 o3 = max(o3tmean2, na.rm = TRUE),
 no2 = median(no2tmean2, na.rm = TRUE))
```

```
## # A tibble: 19 x 4
##     year  pm25    o3   no2
##    <dbl> <dbl> <dbl> <dbl>
##  1  1987 NaN    63.0  23.5
##  2  1988 NaN    61.7  24.5
##  3  1989 NaN    59.7  26.1
##  4  1990 NaN    52.2  22.6
##  5  1991 NaN    63.1  21.4
##  6  1992 NaN    50.8  24.8
##  7  1993 NaN    44.3  25.8
##  8  1994 NaN    52.2  28.5
##  9  1995 NaN    66.6  27.3
## 10  1996 NaN    58.4  26.4
## 11  1997 NaN    56.5  25.5
## 12  1998  18.3  50.7  24.6
## 13  1999  18.5  57.5  24.7
## 14  2000  16.9  55.8  23.5
## 15  2001  16.9  51.8  25.1
## 16  2002  15.3  54.9  22.7
## 17  2003  15.2  56.2  24.6
## 18  2004  14.6  44.5  23.4
## 19  2005  16.2  58.8  22.6
```

summarize() returns a data frame with year as the first column, and then the annual averages of pm25, o3 and no2.

The pipeline operator %>% is very handy for stringing together multiple *dplyr* functions in a sequence of operations. Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e.

third(second(first(x)))

This nesting is not a natural way to think about a sequence of operations. The %>% operator allows you to string operations in a left-to-right fashion, i.e.

first(x) %>% second %>% third

This way we don't have to create a set of temporary variables along the way or create a massive nested sequence of function calls. Once you travel down the pipeline with %>%, the first argument is taken to be the output of the previous element in the pipeline.

R for data analysis

The 3 Steps of Data Analysis

1. Reading in data

- read.table(), read.csv(), read.delim()
- RStudio interface

2. Analysis

- Manipulating & reshaping the data
- any math you like
- plotting the outcome

3. Writing out results

- write.table()
- write.csv()

Example: Copy Number by FISH

10 neuroblastoma patiens were tested for NMYC gene copy number by interphase nuclei FISH. The Amplification of NMYC correlates with worse prognosis. The count data are in a table, with the numbers of cells per patient assayed. For each we have NYMC copy number relative to base ploidy. We need to determine which patients have amplifications (i.e., >25% of cells show NMYC amplification)

The data is a tab delimited text file. Each row is a record, each column is a field. Columns are separated by tabs in the text.

We first need to read in the results table and assign it to an object (rawData).

```
rawData <- read.delim("3_CN_by_FISH.tsv")
# View the first 10 rows to ensure import is OK
rawData[1:10, ] # note data frame contains a patient index column
```

```
##    Patient Nuclei NB_Amp NB_cor NB_Del
## 1        1     42      0     34      8
## 2        2     40      3     30      7
## 3        3     56      6     50      0
## 4        4     42      5     37      0
## 5        5     32      1     30      1
## 6        6     70     10     53      7
## 7        7     65      3     58      4
## 8        8     40      4     31      5
## 9        9     60      0     54      6
## 10       10      61       0      57       4
```

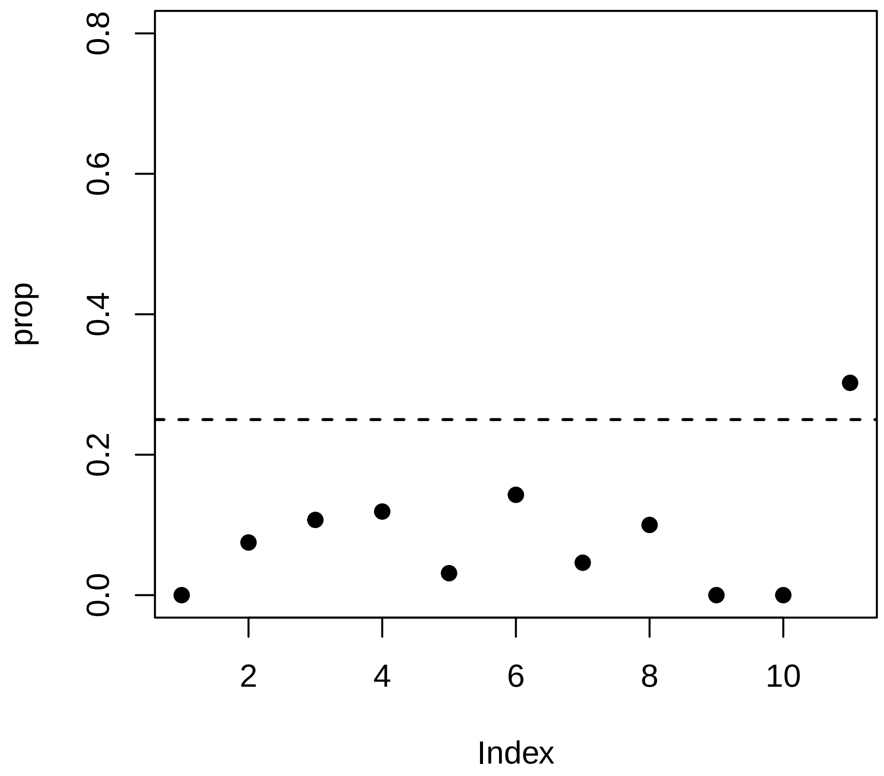
The analysis involves identifying patients with >33% NB amplification.

```
# cretae a vector fraction of cells amplified
prop <- rawData$NB_Amp / rawData$Nuclei
# get row numbers of amplified patients
amp <- prop > 0.25
rawData$Patient[amp]
```

```
## [1] 11
```

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

```
plot(prop, ylim = c(0, 0.8))
abline(h = 0.25, lwd= 1.5, lty = 2)
```



We write out a data frame of results (patients >25% NB amplification) as a comma separated values text file.

```
#export table, file name = selectedSamples.csv
write.csv(rawData[amp, ], file = "selectedSamples.csv")
```

```
# remove the index column of the data frame
rawData <- rawData[, -1]
# use the order function to re-order the data frame by decreasing nuclei count
rawData <- rawData[order(rawData[, 1], decreasing = TRUE), ]
rawData
```

```
##    Nuclei NB_Amp NB_cor NB_Del
## 6      70     10     53      7
## 7      65      3     58      4
## 10     61      0     57      4
## 9      60      0     54      6
## 3      56      6     50      0
## 11     43     13     29      1
## 1      42      0     34      8
## 4      42      5     37      0
## 2      40      3     30      7
## 8      40      4     31      5
## 5      32      1     30      1
```

```
# Patients are near normal if fewer than 25% of the cells are amplified and there are
# zero cells with deletions
prop <- rawData$NB_Amp / rawData$Nuclei
norm <- which( prop < 0.25 & rawData$Del == 0)
norm
```

```
## integer(0)
```

Basic R'Built-in' Functions for Working with Data Objects

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

```
x = sample(100, 15)
```

Arithmetic with vectors

```
# min/max value numbers in a series
min(x) ; max(x)
```

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

```
## [1] 88
```

```
# sum of values in a series
sum(x)
```

```
## [1] 557
```

```
# average estimates (mean/median)
mean(x) ; median(x)
```

```
## [1] 37.13333
```

```
## [1] 38
```

```
# range of values in a series
range(x)
```

```
## [1]  2 88
```

```
# variance
var(x)
```

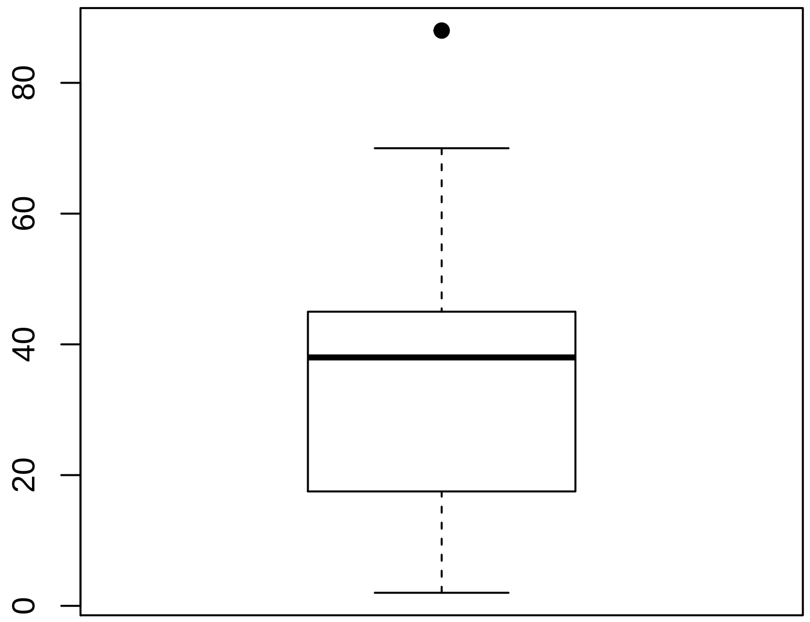
```
## [1] 558.6952
```

```r
# rank ordering
rank(x)
```

```
##  [1]  8 11 12  3  5  2 13 10 14  1  6  9 15  4  7
```

```r
# quantiles
quantile(x) ; boxplot(x)
```

```
##   0%  25%  50%  75% 100% 
##  2.0 17.5 38.0 45.0 88.0
```



```r
# square root
sqrt(x)
```

```
##  [1] 6.164414 6.480741 6.928203 3.741657 4.242641 3.605551 8.062258
##  [8] 6.403124 8.366600 1.414214 5.477226 6.324555 9.380832 4.123106
## [15] 5.567764
```

```r
# standard deviation
sd(x)
```

```
## [1] 23.63673
```

```r
# trigonometry functions
tan(x) ; cos(x) ; sin(x)
```

```
##  [1]  0.3103097  2.2913880  1.2001272  7.2446066 -1.1373137  0.4630211
##  [7] -1.4700383  0.1606567  1.2219599 -2.1850399 -6.4053312 -1.1172149
## [13]  0.0354205  3.4939156 -0.4416956
```

```
##  [1]  0.9550736 -0.3999853 -0.6401443  0.1367372  0.6603167  0.9074468
##  [7] -0.5624539 -0.9873393  0.6333192 -0.4161468  0.1542514 -0.6669381
## [13]  0.9993733 -0.2751633  0.9147424
```

```
##  [1]  0.2963686 -0.9165215 -0.7682547  0.9906074 -0.7509872  0.4201670
##  [7]  0.8268287 -0.1586227  0.7738907  0.9092974 -0.9880316  0.7451132
## [13]  0.0353983 -0.9613975 -0.4040376
```

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

```
names(patients)
```

```
## [1] "First_Name"  "Second_Name" "Full_Name"   "Sex"         "Age"
## [6] "Weight"      "Consent"
```

```
colnames(patients)
```

```
## [1] "First_Name"  "Second_Name" "Full_Name"   "Sex"         "Age"
## [6] "Weight"      "Consent"
```

```
rownames(patients)
```

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

```
# get the numbers of rows or columns with nrow and ncol.
nrow(patients)
```

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

```
ncol(patients)
```

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

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

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

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

```
length(patients)
```

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

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

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

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

You can add rows or columns to a data frame using *rbind* and *cbind*:

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

```
##    First_Name Second_Name      Full_Name    Sex Age Weight Consent
## 1        Adam       Jones     Adam Jones   Male  50   70.8    TRUE
## 2         Eve      Parker     Eve Parker Female  21   67.9    TRUE
## 3        John       Evans     John Evans   Male  35   75.3   FALSE
## 4        Mary       Davis     Mary Davis Female  45   61.9    TRUE
## 5       Peter       Baker    Peter Baker   Male  28   72.4   FALSE
## 6        Paul     Daniels   Paul Daniels   Male  31   69.9   FALSE
## 7      Joanna     Edwards Joanna Edwards Female  42   63.5   FALSE
## 8     Matthew       Smith  Matthew Smith   Male  33   71.5    TRUE
## 9       David     Roberts  David Roberts   Male  57   73.2   FALSE
## 10      Sally      Wilson   Sally Wilson Female  62   64.8    TRUE
## 11       Kate      Lawson    Kate Lawson Female  35   62.5    TRUE
```

```r
# add column with decreasing numbers 10 to 1
cbind(patients, 10:1)
```

```
##    First_Name Second_Name      Full_Name    Sex Age Weight Consent 10:1
## 1        Adam       Jones     Adam Jones   Male  50   70.8    TRUE   10
## 2         Eve      Parker     Eve Parker Female  21   67.9    TRUE    9
## 3        John       Evans     John Evans   Male  35   75.3   FALSE    8
## 4        Mary       Davis     Mary Davis Female  45   61.9    TRUE    7
## 5       Peter       Baker    Peter Baker   Male  28   72.4   FALSE    6
## 6        Paul     Daniels   Paul Daniels   Male  31   69.9   FALSE    5
## 7      Joanna     Edwards Joanna Edwards Female  42   63.5   FALSE    4
## 8     Matthew       Smith  Matthew Smith   Male  33   71.5    TRUE    3
## 9       David     Roberts  David Roberts   Male  57   73.2   FALSE    2
## 10      Sally      Wilson   Sally Wilson Female  62   64.8    TRUE    1
```

You can also remove rows and columns:

```r
# df[-1, ] # remove first row
# df[, -1] # remove first column
```

Sorting a vector with *sort*:

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

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

Sorting a data frame by one variable with *order*:

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

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

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

```
##    First_Name Second_Name      Full_Name    Sex Age Weight Consent
## 5       Peter       Baker    Peter Baker   Male  28   72.4   FALSE
## 6        Paul     Daniels   Paul Daniels   Male  31   69.9   FALSE
## 4        Mary       Davis     Mary Davis Female  45   61.9    TRUE
## 7      Joanna     Edwards Joanna Edwards Female  42   63.5   FALSE
## 3        John       Evans     John Evans   Male  35   75.3   FALSE
## 1        Adam       Jones     Adam Jones   Male  50   70.8    TRUE
## 2         Eve      Parker     Eve Parker Female  21   67.9    TRUE
## 9       David     Roberts  David Roberts   Male  57   73.2   FALSE
## 8     Matthew       Smith  Matthew Smith   Male  33   71.5    TRUE
## 10      Sally      Wilson   Sally Wilson Female  62   64.8    TRUE
```

the R Workspace

The objects we have been making are created in the R workspace. When we load a package, we are loading tht package's functions and data sets into our workspace.

You can see what is in your workspace with *ls* or the Environment tab: *ls()*. You can remove objects from the workspace with *rm*.

Control Structures

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some "logic" into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.

Commonly used control structures are: * **if** and **else**: testing a conditon and acting on it * **for**: execute a loop a fixed number of times * **while**: execute a loop *while* a conditon is true * **repeat**: execute an infinite loop (must break out of it to stop) * **break**: break the execution of a loop * **next**: skip an interaction of a loop

Loops

Many programming languages have ways of doing the same thing many times, perhaps changing some variable each time. This is called *looping*. Loops are not used in R so often, because we can usually achieve the same thing using vector calculations. For example, to add two vectors together, we do not need to add each pair of elements one by one, we can just add the vectors. But there are some situations where R functions can not take vectors as input. For example, *read.csv* will only load one file at a time.

R has two basic types of loop: * **for** loop: run some code on every value in a vector * **while** loop: run some code while some condition is true

```r
for (f in 1:10) {
  print(f)
}
```

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

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

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

If. . . Else. . .

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

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

A condition is any logical value, and can contain multiple conditions (a == b & b < 5).

Statistics in R

R is a statistical programming language. The classical statistical tests are built-in, statistical modeling functions are built-in, Regression analysis is fully supported and additional mathematical packages are available.

Random Numbers and Distributions

The 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

This is availble for variety of distributions: punif (uniform), pbinom (binomial), pnbinom (negative binomial), ppois (poisson), pgeom (geometric), phyper (hypergeometric), pt (T distribution), pf (F distribution) etc.

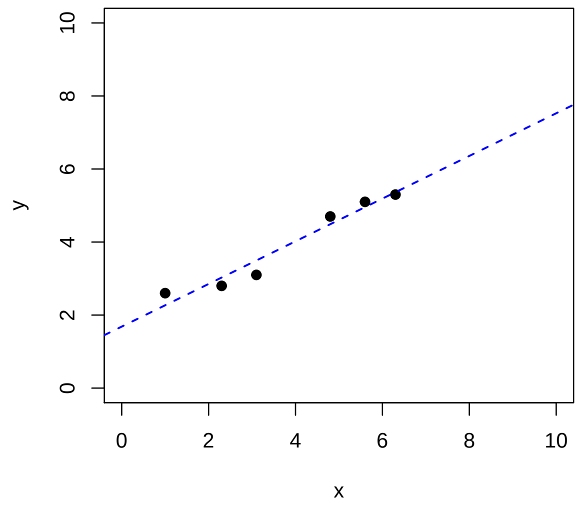
Two sample tests

- Comparing 2 variances - Fisher's F test - var.test()
- Comparing 2 sample means with normal errors - Student's t test - t.test()
- Comparing 2 means with non-normal errors - Wilcoxon rank test - wilcox.test()
- Correlating 2 variables - Pearson/Spearman's rank correlation - cor.test()
- Testing for independence of 2 variables in a contingency table - Chi-squared - chisq.test()
- Fisher's exact test - fisher.test()

Linear Regression

Linear modeling is supported by the function *lm*. *lm* is useful for plotting lines of best fit to XY data in order to determine, intercept, gradient & Pearson's correlaton coefficient.

```r
x <- c(1.0, 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))

myModel <- lm(y ~ x)
abline(myModel, lty = 2, lwd = 1.5, col = "blue")
```



```r
# get the coefficients of the fit form
summary.lm(myModel)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##        1        2        3        4        5        6
##  0.33159 -0.22785 -0.39520  0.21169  0.14434 -0.06458
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.68422    0.29056   5.796   0.0044 **
## x            0.58418    0.06786   8.608   0.0010 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3114 on 4 degrees of freedom
## Multiple R-squared:  0.9488, Adjusted R-squared:  0.936
## F-statistic:  74.1 on 1 and 4 DF,  p-value: 0.001001
```

```r
coef(myModel)
```

```
## (Intercept)           x
##   1.6842239   0.5841843
```

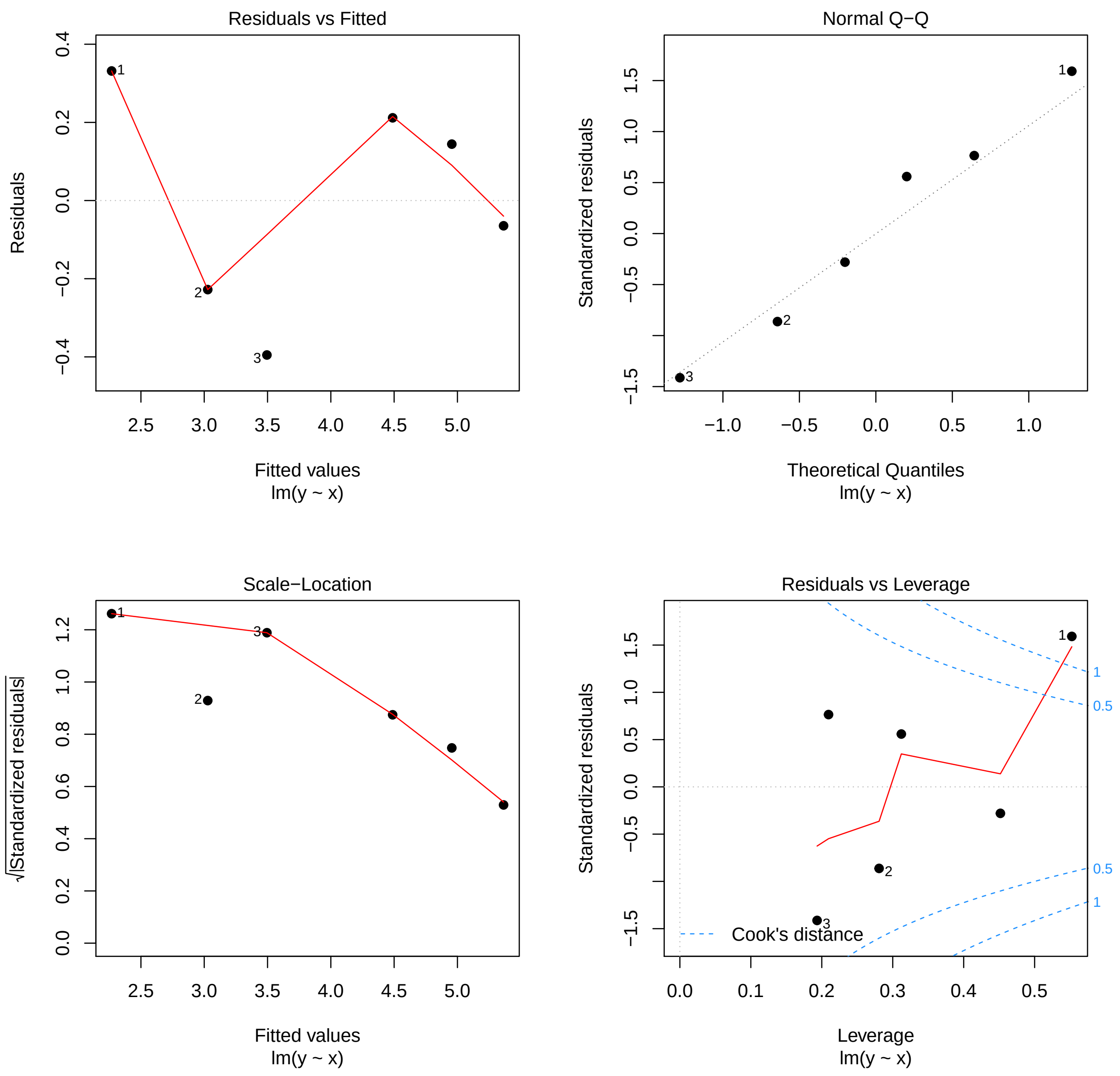
```r
resid(myModel)
```

```
##           1           2           3           4           5           6
##  0.33159186 -0.22784770 -0.39519512  0.21169160  0.14434418 -0.06458482
```

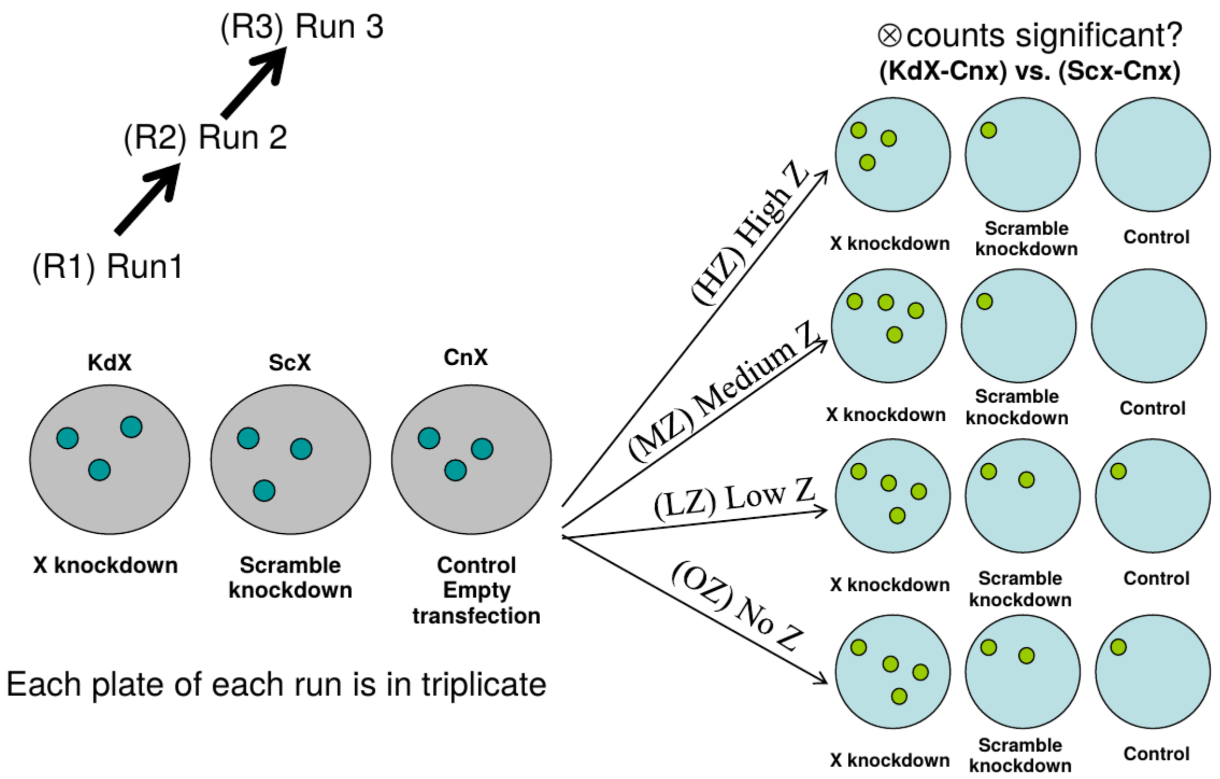
```r
fitted(myModel)
```

```
##        1        2        3        4        5        6
## 2.268408 3.027848 3.495195 4.488308 4.955656 5.364585
```

```r
plot(myModel)
```



Experimental Design



```r
# load in the data from the three runs into three separate data frames t1, t2, t3
t1 <- read.csv("Run1counts.csv")
t2 <- read.csv("Run2counts.csv")
t3 <- read.csv("Run3counts.csv")
# concatenate the three data frames into a single data frame
colony <- rbind(t1, t2, t3)
```

Data is by default read in as factors, i.e. all input strings are enumerated and stored as numbers. The three separate data frame have no indication of which number they came from. We will add a column indication this:

```r
# add the missing Run colum - factors are stored as numbers
runNum <- factor(rep(1:3, each = 36), labels = c("Run1", "Run2", "Run3"))
colony <- cbind("Run" = runNum, colony)

# reorder factor levels in their natural order instead of alphabetical
colony$Treatment <- factor(colony$Treatment, c("OZ", "LZ", "MZ", "HZ"))
colony$Plate <- factor(colony$Plate, c("KDX", "SCX", "CNX"))

# show the full table
colony
```

```
##       Run Run Plate Treatment Replicate Count
## 1    Run1  R1   KDX        OZ         1   100
## 2    Run1  R1   KDX        OZ         2   104
## 3    Run1  R1   KDX        OZ         3    91
## 4    Run1  R1   KDX        LZ         1    85
## 5    Run1  R1   KDX        LZ         2    71
## 6   Run1  R1   KDX        LZ         3    59
## 7   Run1  R1   KDX        MZ         1    32
## 8   Run1  R1   KDX        MZ         2    25
## 9   Run1  R1   KDX        MZ         3    45
## 10  Run1  R1   KDX        HZ         1     5
## 11  Run1  R1   KDX        HZ         2    11
## 12  Run1  R1   KDX        HZ         3     3
## 13  Run1  R1   SCX        OZ         1   136
## 14  Run1  R1   SCX        OZ         2    99
## 15  Run1  R1   SCX        OZ         3   154
## 16  Run1  R1   SCX        LZ         1    79
## 17  Run1  R1   SCX        LZ         2    45
## 18  Run1  R1   SCX        LZ         3    35
## 19  Run1  R1   SCX        MZ         1    15
## 20  Run1  R1   SCX        MZ         2    17
## 21  Run1  R1   SCX        MZ         3    23
## 22  Run1  R1   SCX        HZ         1    49
## 23  Run1  R1   SCX        HZ         2    35
## 24  Run1  R1   SCX        HZ         3    50
## 25  Run1  R1   CNX        OZ         1    23
## 26  Run1  R1   CNX        OZ         2    21
## 27  Run1  R1   CNX        OZ         3    35
## 28  Run1  R1   CNX        LZ         1    32
## 29  Run1  R1   CNX        LZ         2    12
## 30  Run1  R1   CNX        LZ         3    43
## 31  Run1  R1   CNX        MZ         1    14
## 32  Run1  R1   CNX        MZ         2    16
## 33  Run1  R1   CNX        MZ         3    19
## 34  Run1  R1   CNX        HZ         1     8
## 35  Run1  R1   CNX        HZ         2     7
## 36  Run1  R1   CNX        HZ         3    10
## 37  Run2  R2   KDX        OZ         1    89
## 38  Run2  R2   KDX        OZ         2   104
## 39  Run2  R2   KDX        OZ         3    91
## 40  Run2  R2   KDX        LZ         1    32
## 41  Run2  R2   KDX        LZ         2    36
## 42  Run2  R2   KDX        LZ         3    34
## 43  Run2  R2   KDX        MZ         1    32
## 44  Run2  R2   KDX        MZ         2    23
## 45  Run2  R2   KDX        MZ         3    45
## 46  Run2  R2   KDX        HZ         1    67
## 47  Run2  R2   KDX        HZ         2    56
## 48  Run2  R2   KDX        HZ         3    64
## 49  Run2  R2   SCX        OZ         1    33
## 50  Run2  R2   SCX        OZ         2    28
## 51  Run2  R2   SCX        OZ         3    32
## 52  Run2  R2   SCX        LZ         1    23
## 53  Run2  R2   SCX        LZ         2    18
## 54  Run2  R2   SCX        LZ         3    23
## 55  Run2  R2   SCX        MZ         1    78
## 56  Run2  R2   SCX        MZ         2    65
## 57  Run2  R2   SCX        MZ         3    89
## 58  Run2  R2   SCX        HZ         1    32
## 59  Run2  R2   SCX        HZ         2    45
## 60  Run2  R2  SCX         HZ          3   39
## 61  Run2  R2  CNX         OZ          1   23
## 62  Run2  R2  CNX         OZ          2   21
## 63  Run2  R2  CNX         OZ          3   19
## 64  Run2  R2  CNX         LZ          1    8
## 65  Run2  R2  CNX         LZ          2    9
## 66  Run2  R2  CNX         LZ          3   10
## 67  Run2  R2  CNX         MZ          1   14
## 68  Run2  R2  CNX         MZ          2   17
## 69  Run2  R2  CNX         MZ          3   18
## 70  Run2  R2  CNX         HZ          1    9
## 71  Run2  R2  CNX         HZ          2    8
## 72  Run2  R2  CNX         HZ          3    6
## 73  Run3  R3  KDX         OZ          1  100
## 74  Run3  R3  KDX         OZ          2  103
## 75  Run3  R3  KDX         OZ          3   99
## 76  Run3  R3  KDX         LZ          1   64
## 77  Run3  R3  KDX         LZ          2   89
## 78  Run3  R3  KDX         LZ          3   80
## 79  Run3  R3  KDX         MZ          1   43
## 80  Run3  R3  KDX         MZ          2   35
## 81  Run3  R3  KDX         MZ          3   30
## 82  Run3  R3  KDX         HZ          1    4
## 83  Run3  R3  KDX         HZ          2    6
## 84  Run3  R3  KDX         HZ          3    8
## 85  Run3  R3  SCX         OZ          1   10
## 86  Run3  R3  SCX         OZ          2    9
## 87  Run3  R3  SCX         OZ          3    9
## 88  Run3  R3  SCX         LZ          1   38
## 89  Run3  R3  SCX         LZ          2   45
## 90  Run3  R3  SCX         LZ          3   32
## 91  Run3  R3  SCX         MZ          1   69
## 92  Run3  R3  SCX         MZ          2   67
## 93  Run3  R3  SCX         MZ          3   56
## 94  Run3  R3  SCX         HZ          1   70
## 95  Run3  R3  SCX         HZ          2   75
## 96  Run3  R3  SCX         HZ          3   65
## 97  Run3  R3  CNX         OZ          1   35
## 98  Run3  R3  CNX         OZ          2   40
## 99  Run3  R3  CNX         OZ          3   45
## 100 Run3  R3  CNX         LZ          1   58
## 101 Run3  R3  CNX         LZ          2   59
## 102 Run3  R3  CNX         LZ          3   60
## 103 Run3  R3  CNX         MZ          1   15
## 104 Run3  R3  CNX         MZ          2   19
## 105 Run3  R3  CNX         MZ          3   20
## 106 Run3  R3  CNX         HZ          1    8
## 107 Run3  R3  CNX         HZ          2   10
## 108 Run3  R3  CNX         HZ          3    8
```

The tapply Function

Assume we have the following data for heights of 5 males and females.

```r
data <- data.frame(gender = c("Male", "Male", "Female", "Female", "Female"),
                              height = c(6, 6.1, 5.8, 6, 5.95))
```

By calling *mean()* on the height column we can get the average of all 5 people, by how do we get average separately for males and females? Here the *tapply()* functions come into play. It applies a function to grouped data.

```r
tapply (data$height, data$gender, mean) # data, groups, function
```

```
##   Female     Male
## 5.916667 6.050000
```

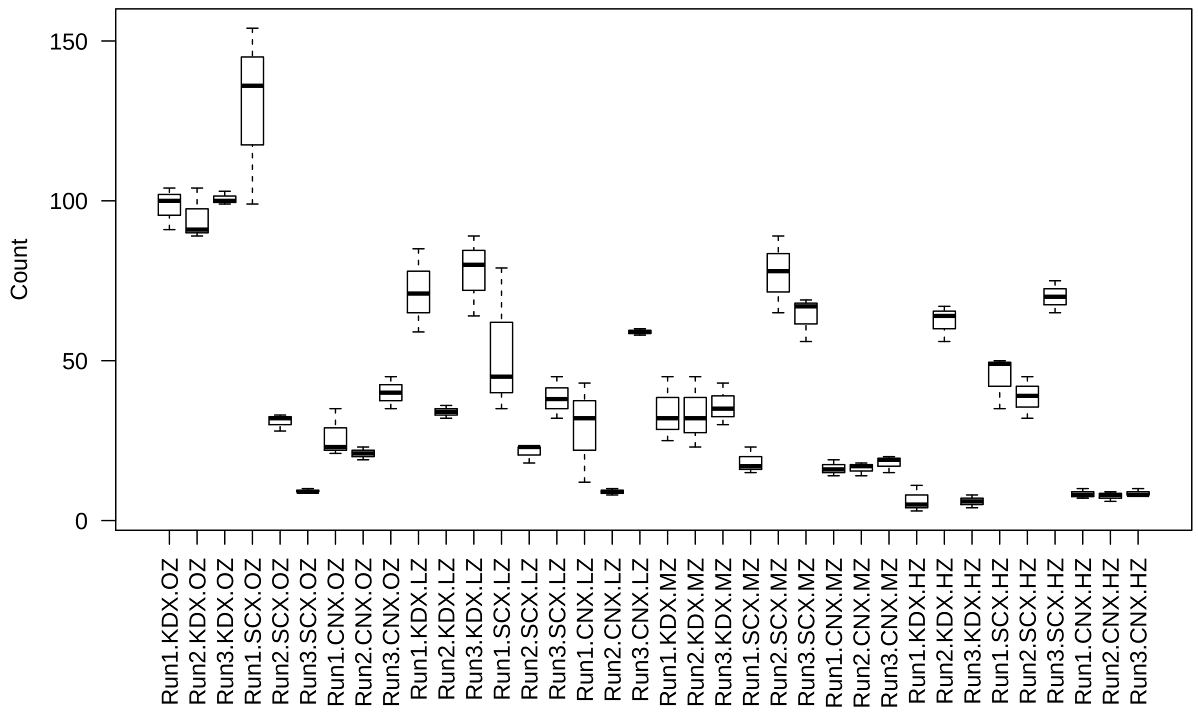
We need the means of the triplicate counts for each run, broken down by plate type (KDX,SCX,CNX) and Z treatment concentration (OZ, LZ, MZ, HZ)

```r
tapply(colony$Count, list(colony$Run, colony$Plate, colony$Treatment), mean)
```

```
## , , OZ
##
##             KDX        SCX      CNX
## Run1  98.33333 129.666667 26.33333
## Run2  94.66667  31.000000 21.00000
## Run3 100.66667   9.333333 40.00000
##
## , , LZ
##
##           KDX      SCX CNX
## Run1 71.66667 53.00000  29
## Run2 34.00000 21.33333   9
## Run3 77.66667 38.33333  59
##
## , , MZ
##
##           KDX      SCX      CNX
## Run1 34.00000 18.33333 16.33333
## Run2 33.33333 77.33333 16.33333
## Run3 36.00000 64.00000 18.00000
##
## , , HZ
##
##            KDX      SCX      CNX
## Run1  6.333333 44.66667 8.333333
## Run2 62.333333 38.66667 7.666667
## Run3  6.000000 70.00000 8.666667
```

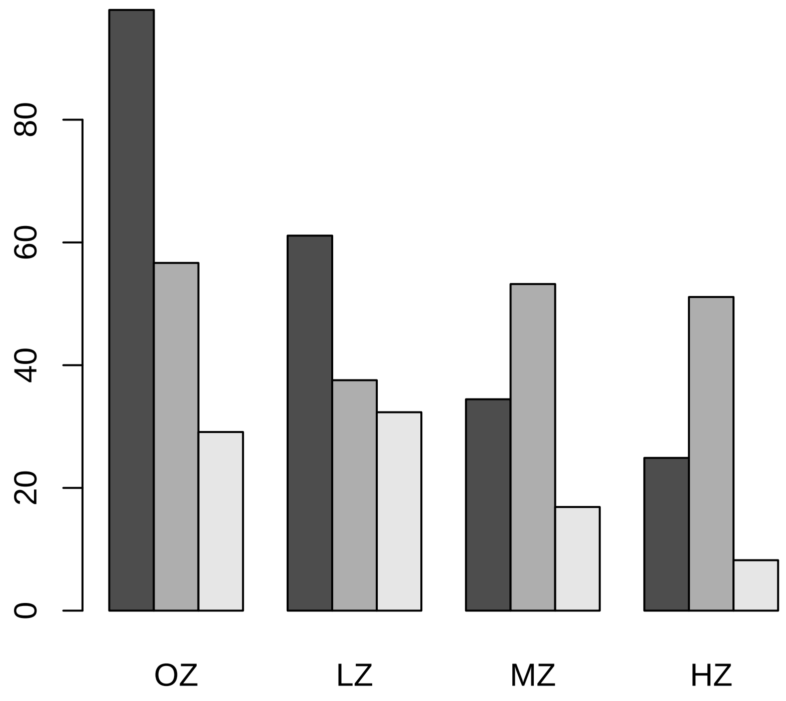
We can plot a graoh of this. It gives us the variation in counts per run

```r
par(oma=c(4,0,4,0))
boxplot(Count ~ Run * Plate * Treatment, las = 2, cex = 0.4, xlab = "", data = colony)
```



Let's plot a grouped bar char of mean counts per plate type per Z treatment

```
barplot(tapply(colony$Count, list(colony$Plate, colony$Treatment), mean), beside = TRUE)
```

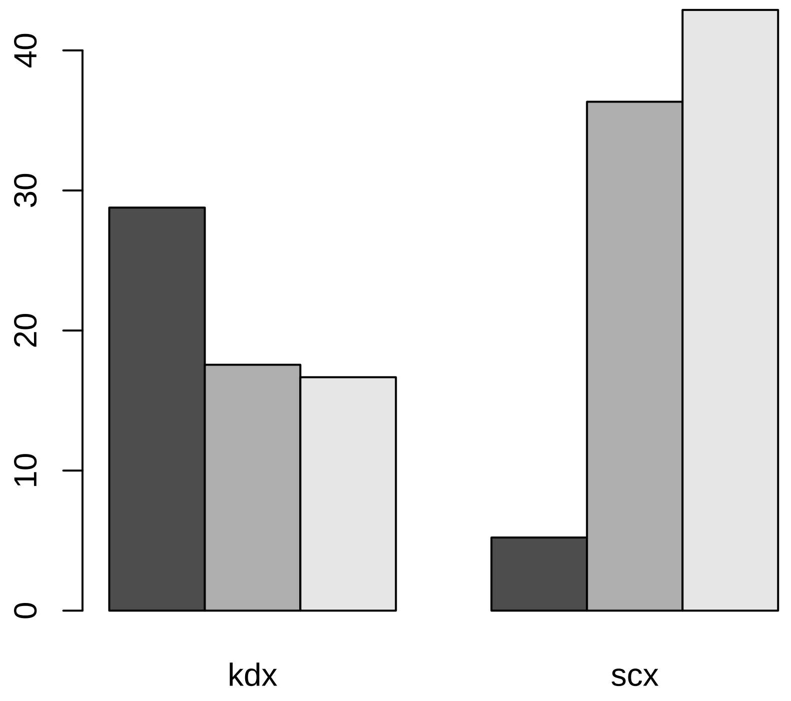


We need a reshaped, background corrected, table of results on which to perform our tests.

```r
result <- tapply(colony$Count, list(colony$Treatment, colony$Plate), mean)
result <- data.frame(result)
result
```

```
##         KDX      SCX       CNX
## OZ 97.88889 56.66667 29.111111
## LZ 61.11111 37.55556 32.333333
## MZ 34.44444 53.22222 16.888889
## HZ 24.88889 51.11111  8.222222
```

```r
# calculate kdx and scx values after background correction
kdx = result$KDX - result$CNX
scx = result$SCX - result$CNX

result <- cbind(kdx, scx)
# remove the OZ entry
result <- result[-1, ]
barplot(result, beside = TRUE)
```



```r
wilcox.test(result[, 1], result[, 2],
paired = TRUE)
```

```
##
##  Wilcoxon signed rank test
##
## data:  result[, 1] and result[, 2]
## V = 2, p-value = 0.75
## alternative hypothesis: true location shift is not equal to 0
```

```r
cor.test(result[, 1], result[, 2],
paired = TRUE)
```

```
##
##  Pearson's product-moment correlation
##
## data:  result[, 1] and result[, 2]
## t = -10.2, df = 1, p-value = 0.06222
## alternative hypothesis: true correlation is not equal to 0
## sample estimates:
##        cor
## -0.9952285
```

```r
write.csv(result, "CFAresults.csv")
```

Functions Introduction

All R commands are functions. Functions perform calculations, possibly involving several arguments, then return a value to the calling statement. The calculation maybe any process, might or might not have return vale, it need not to be arithmetic. User functions extend the capabilities of R by adapting or creating new tasks that are tailored to your specific requirements. User functions are a special kind of object.

```
myfunction <- function(x) {
  return(x^2 + x)
}
myfunction(10)
```

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

Example of function with more than one argument:

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

Now we have two arguments. The second argument has a default value of 2. Arguments without default value are required, those with default values are optional. Functions may be assigned a name, or anonymously created within an operation. Anonymous functions are really useful in *apply()* style procedures.

apply(object, margin, function)

```
a <- matrix(1:10, ncol = 10, byrow = TRUE)
a
```

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

```
apply(a, c(1,2), function(x) x^2 +x)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    2    6   12   20   30   42   56   72   90   110
```

Objects created in functions are not available to the general environment unless returned. They are said to be "out of scope". Scope relates to the accessibility of an object. A function can only return one object. Custom functions disappear when R sessions end, unless the function object is saved in an Rdata file or sourced from a script. A really useful function could be added to your .Rprofile file, and would always be ready for you at launch. If your script repeats the same style command more than twice, you should consider writing a function. Writing functions makes your code more easily understandable because they encapsulate a procedure into a well-defined boundary with consistent input/output. Functions should not be longer than one-to-two screens of code, keep functions clean and simple. Look at other functions to get ideas for how to write your own. Display function code by entering the functions name withount brackets.

Example: Temperature Conversion Function Centigrade to Fahrenheit conversion is given by F=9/5 C + 32. Write a function that converts between temperatures. The function will need two named arguments with default values.

```
# convTemp is defined as a new user function requiring two arguments, temp and unit
# the default values are 0 and "c".

convTemp <- function(temp=0, unit="c") {
  if (!is.numeric(temp)){
    stop("Non numeric temperature entered") # Exception error if character given for temp
  }
  if(!(unit=="c" | unit=="f")){
    stop("Unrecognized temperature unit. \n Enter either (c)entigrade, or
```

```r
         (f)ahrenheit") # Exception for unit error
  }
  # conversion for centrigrade
  if (unit=="c"){
    fTemp <- 9/5 * temp + 32
    output <- paste(temp, "C is: ", fTemp, "F \n")
    cat(output)
  }
  # conversion for fahrenheit
  if (unit=="f"){
    cTemp <- 5/9 * (temp-32)
    output <- paste(temp, "F is: ", cTemp, "C \n")
    cat(output)
  }
}

convTemp(37)
```

```
## 37 C is:  98.6 F
```

```r
convTemp(100, unit="f")
```

```
## 100 F is:  37.7777777777778 C
```

```r
convTemp(100, unit="c")
```

```
## 100 C is:  212 F
```

```r
convTemp(unit="f", 314)
```

```
## 314 F is:  156.666666666667 C
```

Loop Functions

- **lapply()**: Loop over a list and evaluate a function on each element
- **sapply()**: Same as *lapply* but try to simplify the result
- **apply()**: Apply a function over the margins of an array
- **tapply**: Apply a function over subsets of a vector
- **mapply**: Multivariate version of *lapply*

lapply()

The *lapply() function does the following simple series of operations: 1. it loops over a list, iterating over each element in that list 2. it applies a* function* to each element of the list (a function that you specify) 3. and returns a list

```r
x <- list(a=1:5, b=rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.1276488
```

```
a
```

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

```
b
```

```
##             [,1]       [,2]      [,3]       [,4]      [,5]      [,6]
##  [1,] 0.95713472 0.12928524 0.9136112 0.08293578 0.1935961 0.8471330
##  [2,] 0.08647524 0.02551505 0.4059887 0.87775360 0.8207337 0.4152601
##  [3,] 0.20041288 0.45539534 0.7225878 0.84768588 0.5095537 0.5961076
##  [4,] 0.43585787 0.03142108 0.1250795 0.75733666 0.1966052 0.7215346
##  [5,] 0.92780652 0.97407967 0.2993291 0.17687231 0.4846386 0.1146248
##  [6,] 0.63246339 0.21205699 0.6396624 0.72315901 0.4525778 0.1006474
##  [7,] 0.47174720 0.78289661 0.6302762 0.37404335 0.3507625 0.0488728
##  [8,] 0.34677824 0.45996496 0.7473380 0.65401325 0.6314462 0.7664913
##  [9,] 0.99155765 0.73369556 0.9105560 0.04716393 0.7707671 0.9743816
## [10,] 0.21103142 0.77821923 0.1242174 0.36792858 0.9856963 0.1683770
##             [,7]        [,8]       [,9]       [,10]
##  [1,] 0.36986015 0.394695625 0.68743691 0.388352681
##  [2,] 0.30890596 0.002870526 0.74040521 0.126163972
##  [3,] 0.37159692 0.066997095 0.05219229 0.793644837
##  [4,] 0.05379184 0.874000326 0.49849840 0.499377728
##  [5,] 0.18624502 0.918682075 0.45136903 0.112493155
##  [6,] 0.96664969 0.241690222 0.86568307 0.255512080
##  [7,] 0.21042192 0.116756669 0.87343620 0.550237532
##  [8,] 0.25261279 0.440952670 0.02806077 0.383245673
##  [9,] 0.57908789 0.579628467 0.48405738 0.120887060
## [10,] 0.04751861 0.678015933 0.91799444 0.006602674
```

Notice that here we are passing the *mean()* function as an argument to the *apply()* function. Functions in R can be used this way and can be passed back and forth as arguments just like any other object. When you pass a function to another function, you do not need to include the open and closed parentheses() like you do when you are *calling* a function.

The *lapply()* function and its friends make heavy use of *anonymous* functions. Anonymous functions have no names. These functions are generated "on the fly" as you are using *lapply()*. Once the call to *lapply()* is finished, the function disappears and does not appear in the workspace.

```
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $b
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
lapply(x, function(elt) {elt[,1]})
```

```
## $a
## [1] 1 2
##
```

```
## $b
## [1] 1 2 3
```

Notice that the *function()* definition is right in the call to *lapply()*. This is perfectly legal and acceptable. You can put arbitrarily complicated function definition inside *lapply()*, but if it's going to be more complicated, it's probably a better idea to define the function separately.

sapply()

The *sapply()* function behaves similarily to *lapply()*; the only real difference is in the return value. *sapply()* will try to simplify the result of *lapply()* if possible. Essentially, *sapply()* calls *lapply()* on its input and then applies the following algorithm: * if the result is a list where every element is lenght 1, then a vector is returned * if the result is a list where every element is a vector of the same length (>1), a matrix is returned. * if it can't figure out, a list returned

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] 0.3709813
##
## $c
## [1] 1.216952
##
## $d
## [1] 5.077892
```

```
sapply(x, mean)
```

```
##         a         b         c         d
## 2.5000000 0.3709813 1.2169523 5.0778920
```

Because the result of *lapply()* was a list where each element had lenght 1, *sapply()* collapsed the output into a numeric vector, which is often more useful than a list.

split()

The *split()* function takes a vector or other object and splits it into groups determined by a factor or list of factors. The combination of *split()* and a function like *lapply* okr *sapply()* is a common paradigm in R. The basic idea is that you can take a data structure, split it into subsets defined by another variable, and apply a function over those subsets. The results of applying that function over the subsets are then collated and returned as an object. This sequence of operations is sometimes referred to as "map-reduce" in other contexts.

```
x <- c(rnorm(10), runif(10), rnorm(10,1))
f <- gl(3, 10)
split(x, f)
```

```
## $`1`
##  [1] -0.04620113 -0.58629251  0.36330513  0.21601634 -0.23211688
##  [6] -1.00447521  0.76386203 -1.46213963 -1.26459474  0.92328694
##
## $`2`
##  [1] 0.36792416 0.57433242 0.44007064 0.03792177 0.98898791 0.23265656
##  [7] 0.92495529 0.29651991 0.06160237 0.09394496
##
## $`3`
##  [1] -0.1680778  2.9759392  0.6610744  0.4600765 -0.6459713 -0.1286370
##  [7]  0.8069836  1.4326577  1.1237309  1.8537630
```

```
lapply(split(x,f), mean)
```

```
## $`1`
## [1] -0.232935
##
## $`2`
## [1] 0.4018916
##
## $`3`
## [1] 0.8371539
```

tapply()

tapply() is used to apply a function over a subsets of a vector. It can be thought of as a combination of *split()* and *sapply()* for vectors only.

```
#  simulate some data
x <- c(rnorm(10), runif(10), rnorm(10.1))
# define some groups with a factor variable
f <- gl(3, 10)
f
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
## Levels: 1 2 3
```

```
tapply(x, f, mean)
```

```
##          1          2          3
##  0.1315639  0.5077019 -0.5209499
```

We can also take the group mean without simplyfying the result, which will give us a list. For functions that return a single value, usually, this is not what we want, but it can be done.

```
tapply(x, f, mean, simplify = FALSE)
```

```
## $`1`
## [1] 0.1315639
##
## $`2`
## [1] 0.5077019
##
## $`3`
## [1] -0.5209499
```

apply()

The *apply()* function is used to evaluate a function (often an anonymous one) over the margins of an array. It is most often used to apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). However, it can be used with general arrays, for example, to take the average of na array of matrices. Using *apply()* is not really faster than writing a loop, but it works in one line and is highly compact.

```r
# createa 20 by 10 matrix of normal random numbers, then compute the mean of each column
x <- matrix(rnorm(200), 20, 10)
apply(x, 2, mean) # take the mean of each column
```

```
##  [1] -0.04807534 -0.13909812  0.40075282 -0.19790947  0.19634864
##  [6] -0.06189601  0.01502404 -0.17914160 -0.33293257 -0.29588807
```

```r
apply(x, 1, sum) # take the mean of each row
```

```
##  [1]  8.2248813 -0.7713607 -1.5656325  1.3023288  2.8457768 -2.3148954
##  [7] -4.3597658  1.3528903  0.2058499 -1.3274471 -5.0271577 -3.8605406
## [13] -3.7516214  2.2605151  3.2971931  2.1892630 -6.6290208 -1.6981535
## [19] -0.8957667 -2.3336496
```

Note that in both calls to *apply()*, the return value was a vector of numbers. You've probably noticed that the second argument is either a 1 or a 2, depending on whether we want row statistics or column statistics. This margin argument essentially indicates to *apply()* which dimension of the array you want to preserve or retain.

Col/Row Sums and Means

For the special case of colulmn/row sums and column/row means of matrices, we have some useful shortcuts. * **rowSums** = apply(x, 1, sum) * **rowMeans** = apply(x, 1, mean) * **colSums** = apply(x, 2, sum) * **colMeans** = apply(x, 2, mean)

mapply()

The *mapply()* function is a multivariate apply of sorts which applies a function in parallel over a set of arguments. Recall that *lapply()* and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what *mapply()* is for.

The *mapply()* function has a different argument order from *lapply()* because the function to apply comes first rather than the object to iterate over. The R objects over which we apply the function are given in the argument because we can apply over an arbitrary number of R objects.

```r
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

Regular Expressions

The primary R functions for dealing with regular expressions are: * **grep(), grepl()**: These functions search for matches of a regular expression/pattern in a character vector. *grep()* returns the indices into the character vector that contain a match or the specific strings hat happen to have the match. *grep1()* returns a TRUE/FALSE vector indicating which elements of the character vector contain a match. * **regexpr(), gregexpr()**: Search a character vector for regular expression matches and return the indices of the string where the match begins and the length of the match * **sub(), gsub()**: Search a character vector for regular expression matches and replace that match with another string * **regexec()**: This function searches a character vector for a regular expression, much like regexpr(), but it will additionally return the locations of any parenthesized sub-expressions.

Simulation

Generating Random Numbers

Simulation is an important topic for both statistics and for a variety of other areas where there is a need to introduce randomness. Sometimes you want to implement a statistical procedure that requires random number generation or sampling and sometimes you want to simulate a system and random number generators can be used to model randon inputs.

R comes with a set of pseudo-random number generators that allow you to simulate from well known probability distributions like the Normal, Poisson, and binomial.

Some example functions for probability distributions in R: * **rnorm**: generate random Normal variates with a given mean and standard deviation * **dnorm**: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points) * **pnorm**: evaluate the cumulative distribution function for a Normal distribution * **rpois**: generate random Poisson variates with a given rate

For each probability distribution there are typically four functions available that start with a "r", "d", "p", and "q". The "r" function is the one that actually simulates random numbers from that distributin. The other functions are prefixed with a

- **d** for density
- **r** for random number generation
- **p** for cumulative distribution
- **q** for quantile function (inverse cumulative distribution)

```
# Simulate standard Normal random numbers
x <- rnorm(10)
x
```

```
##  [1]  1.3151140 -0.4358609  0.8312605  0.5177201 -1.0190445 -0.8765304
##  [7] -1.1829709  1.1205200  0.7484625  1.1563840
```

```
# modify the default parametes to simulate numbers with mean 20 and standard deviation 2.
x <- rnorm(10,20,2)
x
```

```
##  [1] 19.68779 21.46468 20.80606 19.07997 19.48732 24.24938 17.67761
##  [8] 16.25106 20.87496 19.25150
```

```
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   16.25   19.12   19.59   19.88   20.86   24.25
```

```
# if you want to know the probability of less than 2
pnorm(2)
```

```
## [1] 0.9772499
```

Setting the random number seed

When simulating any random numbers it is essential to set the *random number seed.* Setting random number seed with set.seed() ensures reproducibility of the sequence of random numbers.

```
set.seed(1)
rnorm(5)
```

```
## [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

```
rnorm(5)
```

```
## [1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
```

```
# if you want to reproduce the original, reset seed with set.seed().
set.seed(1)
rnorm(5)
```

```
## [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

In general, you should **always set the random number seed when conducting a simulation!**. Otherwise, you will not be able to reconstruct the exact numbers that you produced in an analysis.

Simulating a Linear Model

Simulating random numbers is useful but sometimes we want to simulate values that come from a specific *model*. For that we need to specify the model and then simulate from it using the functions described above.

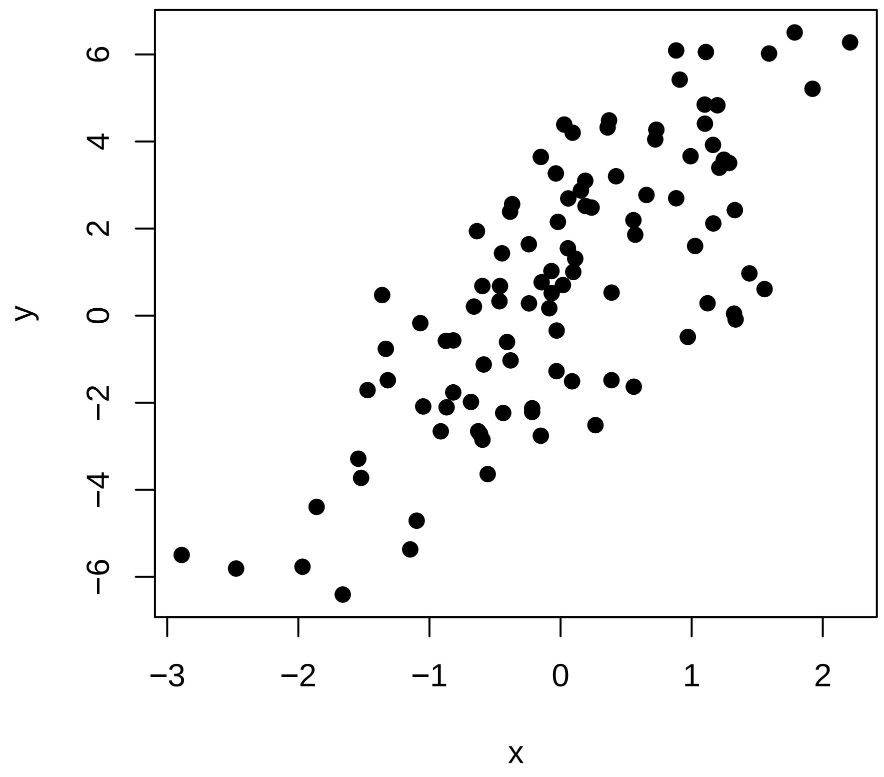
Suppose we want to simulate from the following linear model

y = beta + beta1*x + epsilon;

```
# Always set your seed!
set.seed(20)
#Simulate predictor variable
x <- rnorm(100)
# Simulate the error term
e <- rnorm(100, 0, 2)
# Compute the outcome via the model

y <- 0.5 + 2 *
x + e
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -6.4084 -1.5402  0.6789  0.6893  2.9303  6.5052
```

```
plot(x, y)
```

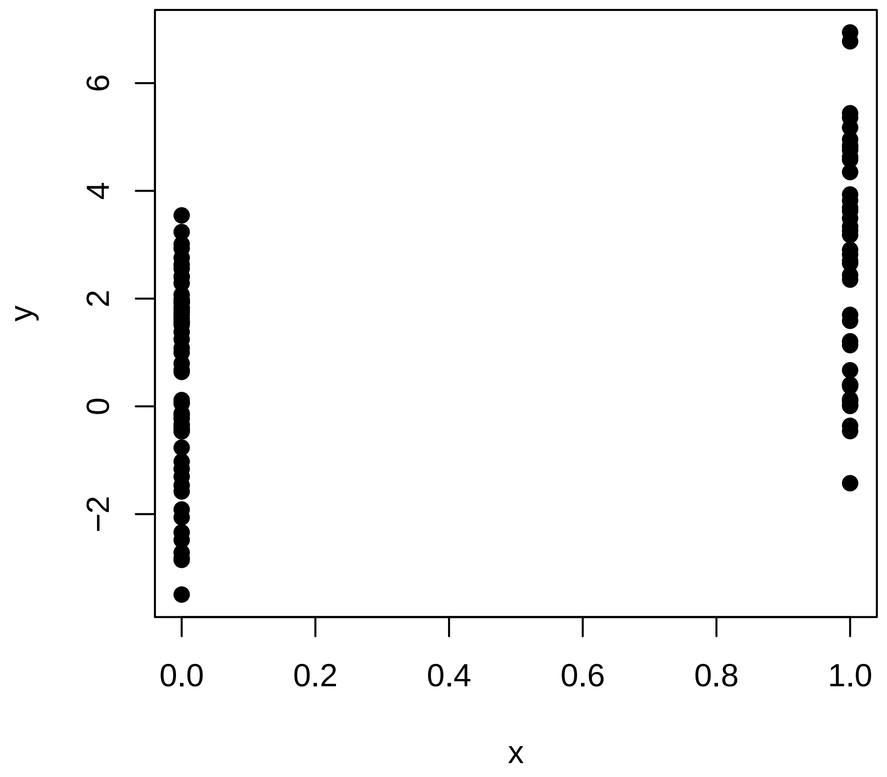


What if we wanted to simulate a predictor variable x that is binary instead of having a Normal distribution. We can use the rbinon() function to simulate binary random variables.

```r
set.seed(10)
x <- rbinom(100, 1, 0.5)
str(x)
```

```
##  int [1:100] 1 0 0 1 0 0 0 0 1 0 ...
```

```r
# 'x' is now 0s and 1s

e <- rnorm(100, 0, 2)
y <- 0.5 + 2 * x + e
plot(x, y)
```

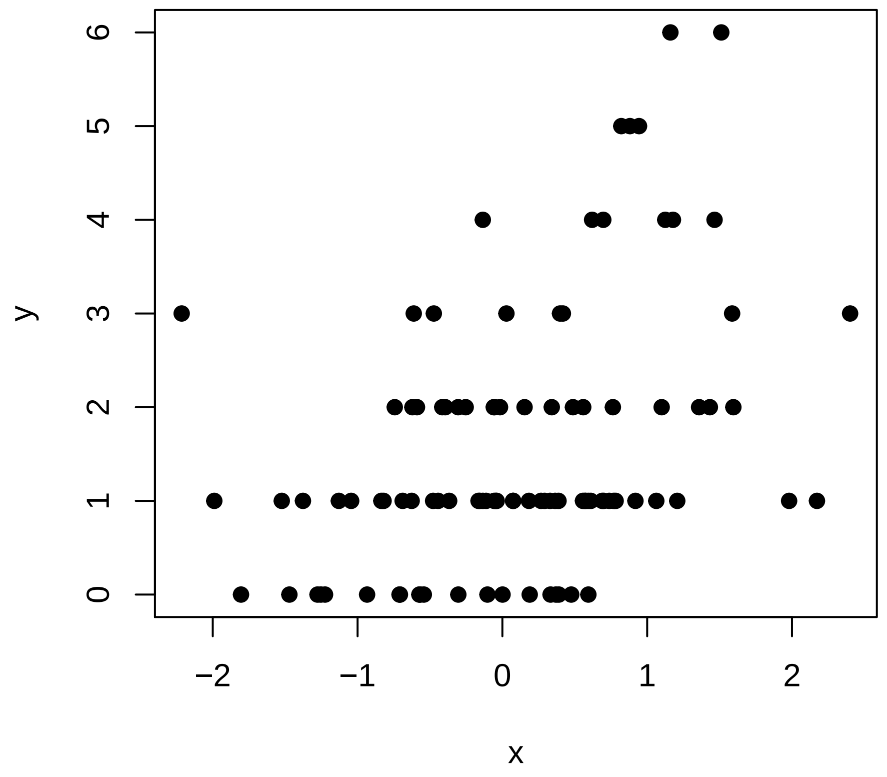


We can also simulate from *generalized linear model* where the errors are no longer from a Normal distrivution but come from some other distribution. For example, suppose we want to simulate from Poisson log-linear model.

```
set.seed(1)
# Simulate the predictor variable as before
x <- rnorm(100)
log.mu <- 0.5 + 0.3 * x
y <- rpois(100, exp(log.mu))
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    1.00    1.00    1.55    2.00    6.00
```

```
plot(x, y)
```



You can build arbitrarily complex models like this by simulating more predictors or making transformations of those predictors (e.g. squaring, log transformation, etc.)

Random Sampling

The *sample()* function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions of numbers.

```
set.seed(1)
sample(1:10, 4)
```

```
## [1] 9 4 7 1
```

```
sample(1:10, 4)
```

```
## [1] 2 7 3 6
```

```
# Doesn't have to be numbers
sample(letters, 5)
```

```
## [1] "r" "s" "a" "u" "w"
```

```
# Do a random permutation
sample(1:10)
```

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

```r
sample(1:10)
```

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

```r
# Sample w/replacement
sample(1:10, replace = TRUE)
```

```
##  [1]  3  6 10 10  6  4  4 10  9  7
```

To sample more complicated things, such as rows from a data frame or a list, you can sample the indices into an object rather than the elements of the object itself.

```r
library(datasets)
data(airquality)
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

Now we just need to create the index vector indexing the rows of the data frame and sample directly from that index vector.

```r
set.seed(20)
# Create index vector
idx <- seq_len(nrow(airquality))

# Sample from the index vector
samp <- sample(idx, 6)
airquality[samp, ]
```

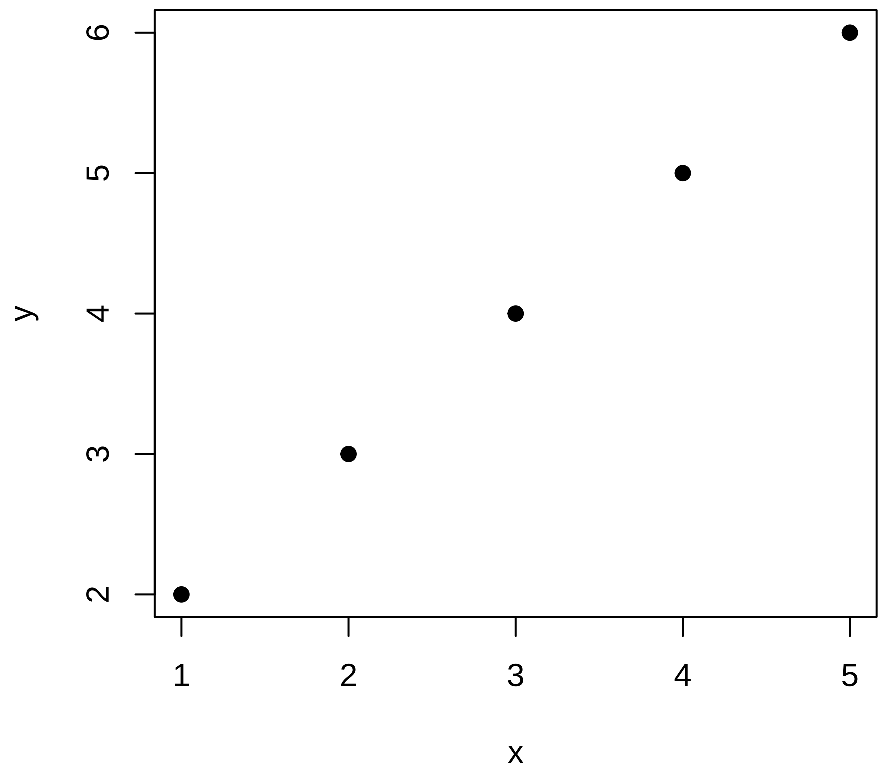
```
##     Ozone Solar.R Wind Temp Month Day
## 107    NA      64 11.5   79     8  15
## 120    76     203  9.7   97     8  28
## 130    20     252 10.9   80     9   7
## 98     66      NA  4.6   87     8   6
## 29     45     252 14.9   81     5  29
## 45     NA     332 13.8   80     6  14
```

R Graphics

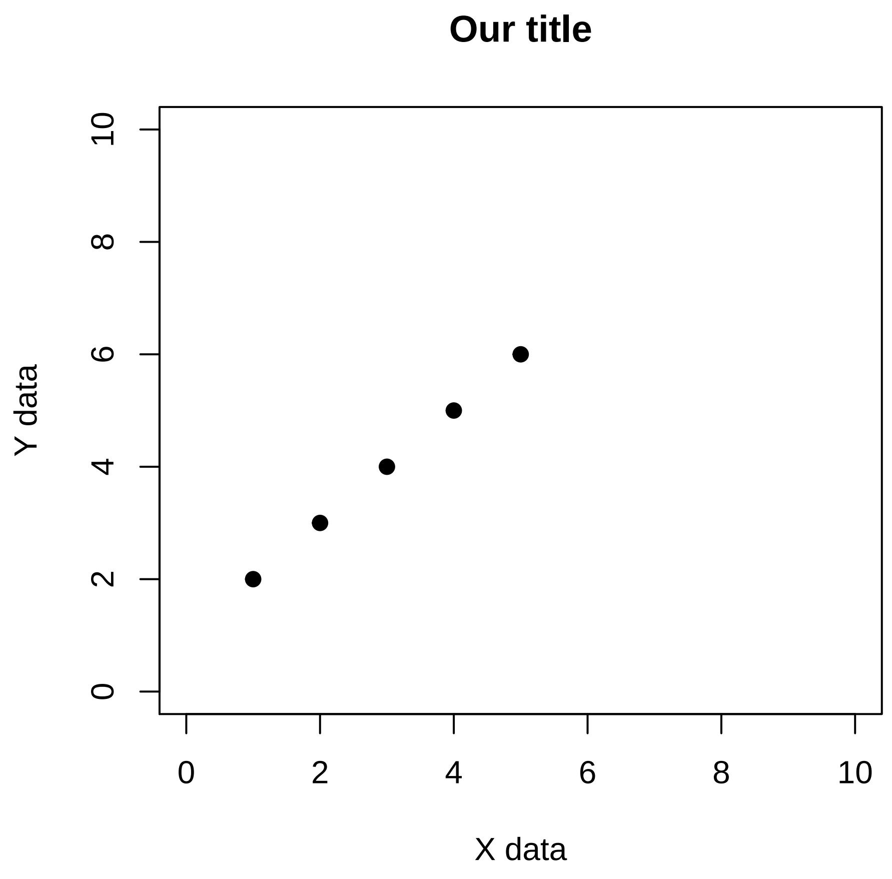
plot()

plot() is the main function for plotting, it takes x,y values to plot and also lots of graphical parameters.

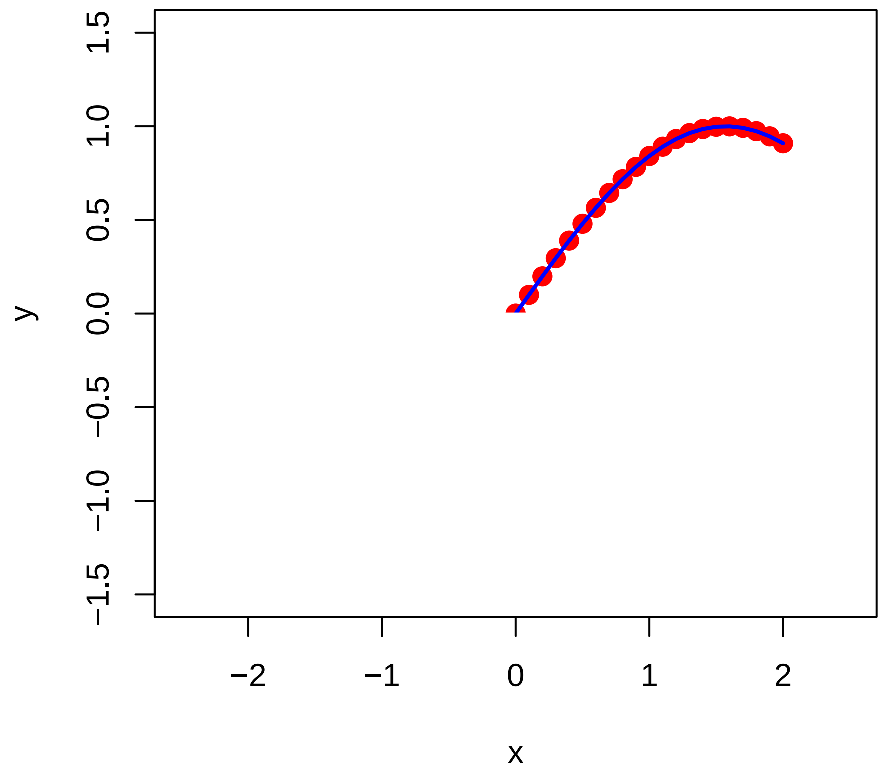
```r
x <- 1:5
y <- 2:6
plot(x, y)
```



```r
plot(x, y, xlab = "X data", # x label
     ylab = "Y data", # y label
     xlim = c(0,10), # x axis limits
     ylim = c(0,10), # y axis limits
     main = "Our title") # title of plot
```

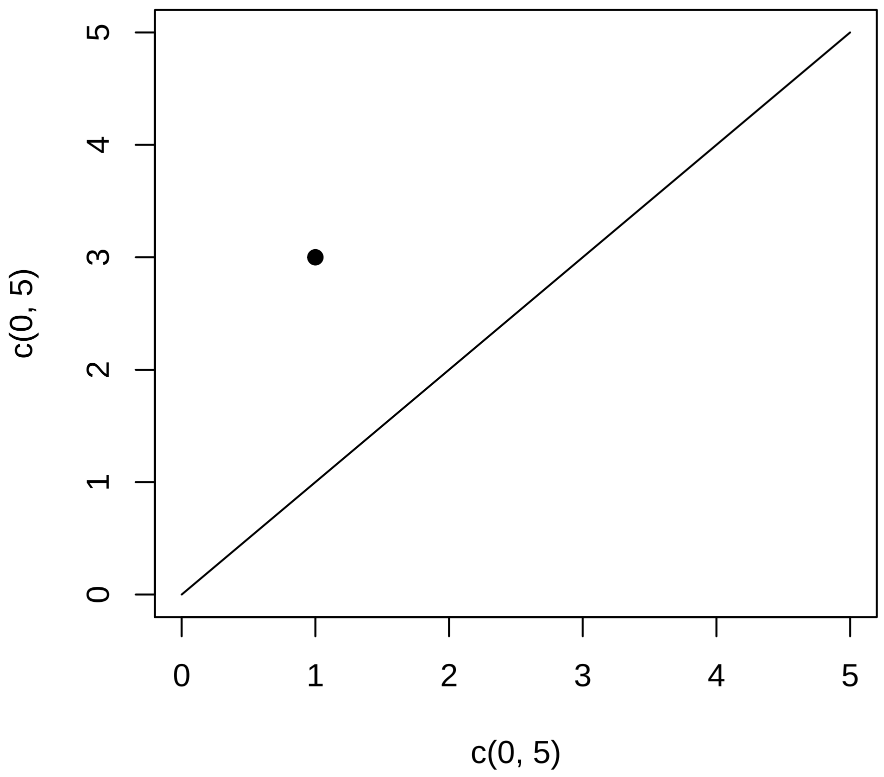


```r
x <- seq(-2, 2, 0.1)
y <- sin(x)

plot(y~x,
     ylim = c(-1.5, 1.5),
     xlim = c(-2.5, 2.5),
     col = "red", # line color
     pch = 16, # plotting chararcter (dot, sqare, triangle etc.)
     cex = 1.4) # character expansion (scaling factor)

# connecting line
lines(y ~ x,
      ylim = c(-1.5, 1.5),
      xlim = c(-2.5, 2.5),
      col = "blue",
      lty = 1, # line type
      lwd = 2) # line width

# draw a rectangle over plot
rect(-2.5, 0, 2.5, -1.5,
     col = "white",
     border = "white")
```



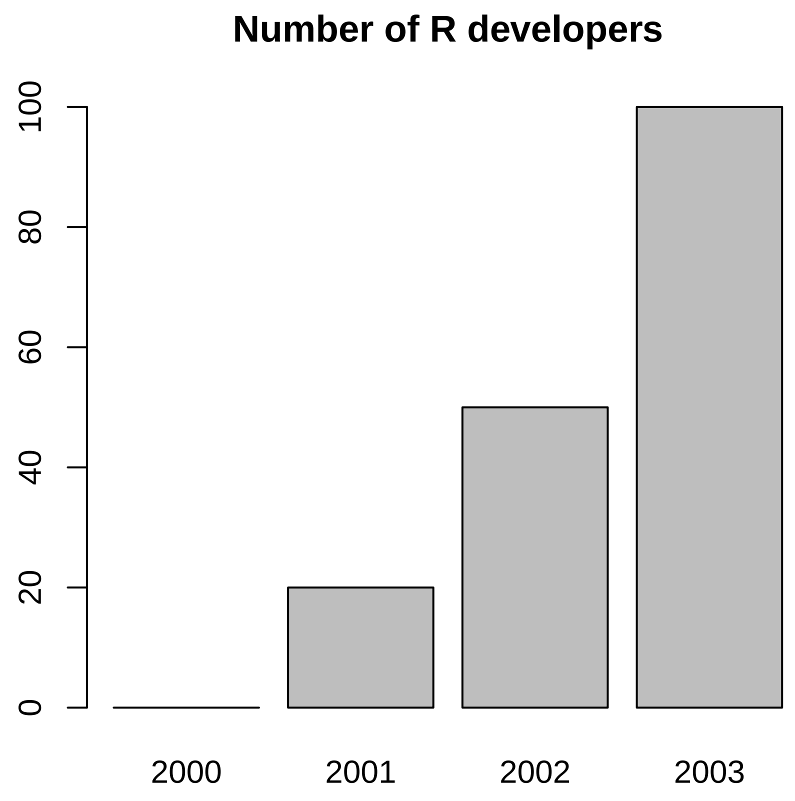
plot() is used to start a new plot, it accepts x,y data, but also data from some objects (like linear regression). *points()* is used to add points to an existing plot *lines()* is used to add lines to an existing plot

```r
# 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)
```



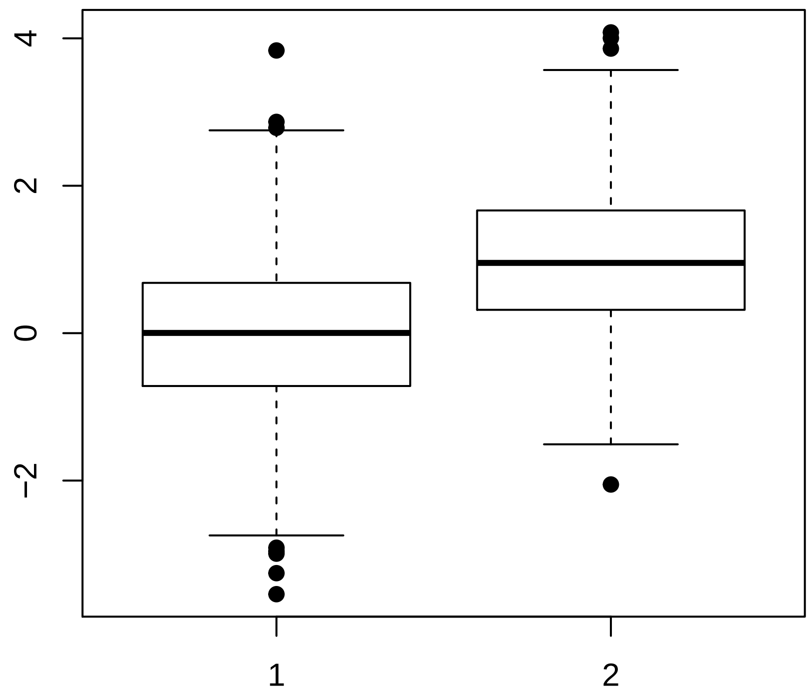
#### barplot() Visualizing a vector of data can be done with bar plots, using function *barplot()*.

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



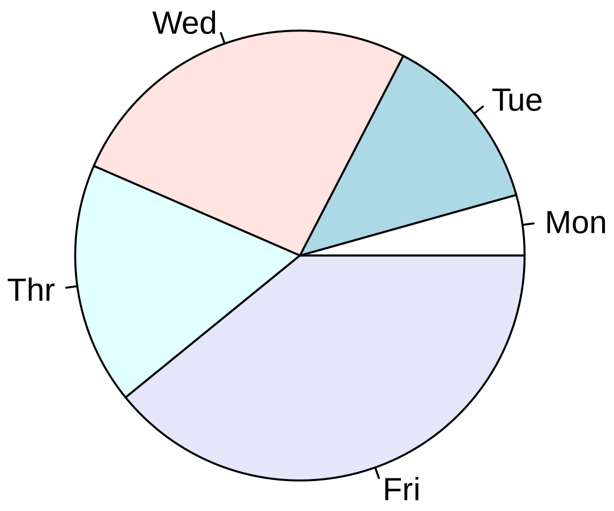
#### boxplot() When a spread of data needs to be visualised, we can use boxplots with function *boxplot()*.

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



#### pie() To visualize percentages or parts of a whole we can use pie charts with *pie()*.

```r
data <- c("Mon" = 1, "Tue" = 3, "Wed" = 6, "Thr" = 4, "Fri" = 9)
pie(data)
```

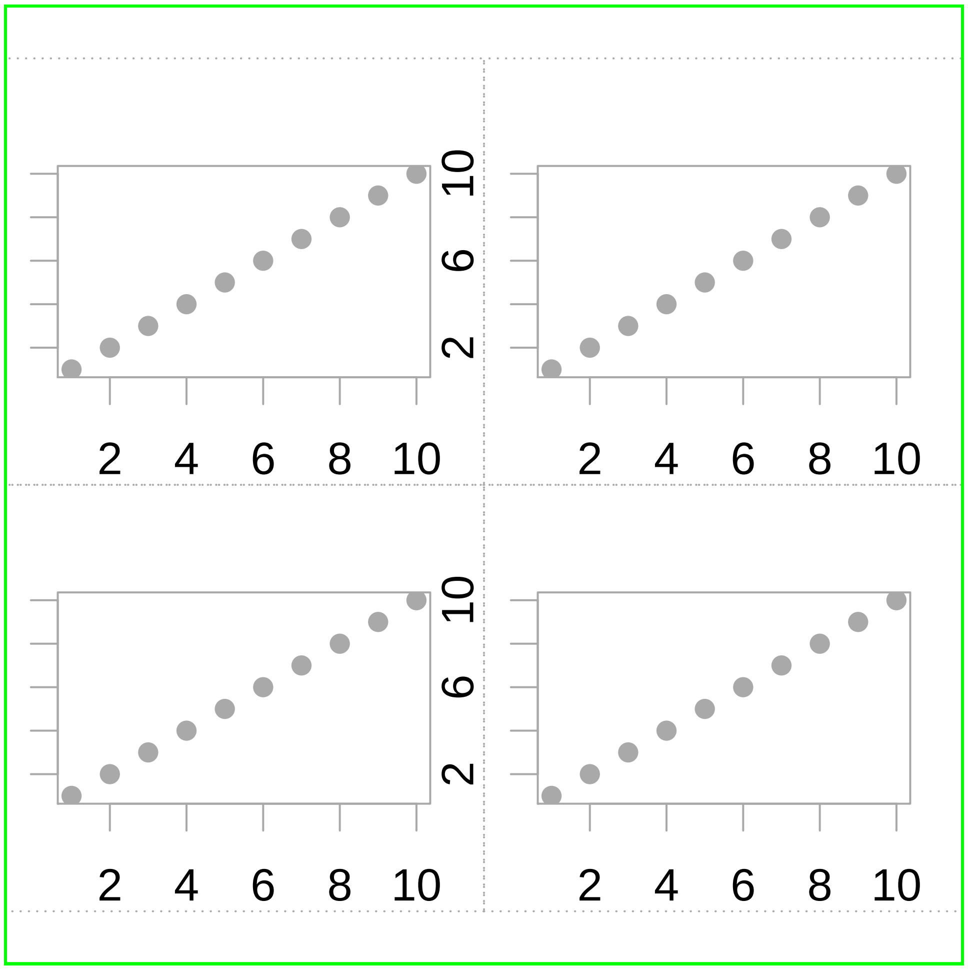


Setting Graphics Layout and Style - par()

To set the global plot layout and style use the par() function. You are able to set the number of plots you want per page, the outer margins of the figure region, the inner margins of the plot and set styles for the plot (colors, line styles and weights).

```r
# 2 x 2 figures per page
par(mfrow = c(2, 2))

# 1 line spacing top and bottom
par(oma = c(1, 0, 1, 0))

# 4 lines at bottom & top
# 2 lines at left & right
par(mar = c(2, 1, 2, 1))

# darkgrey foreground
par(fg = "darkgrey")

# large circles for spots
par(pch = 16,cex = 1.4)

plot(1:10)
```

```r
box("figure", lty = 3)

plot(1:10)
box("figure", lty = 3)

plot(1:10)
box("figure", lty = 3)

plot(1:10)
box("figure", lty = 3)

# draw a green solid line around figure
box("outer", lty = 1, lwd = 3, col = "green")
```



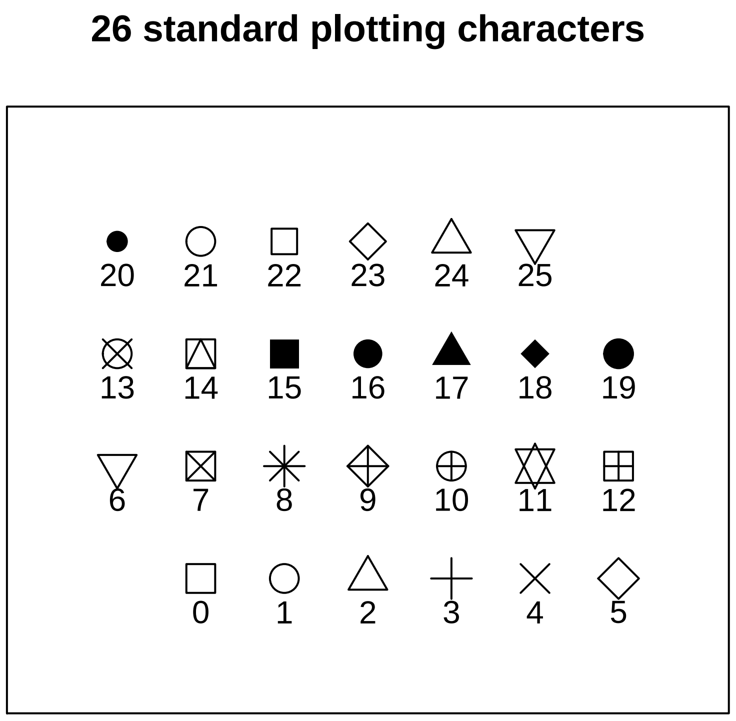
pch=... sets one of the 26 standard plotting character used.

cex=... character expansion, sets the scaling factor of the printing character

las=... axes label style. 1 normal, 2 rotated 90 degrees.

```r
xCounter <- 1
yCounter <- 1
plotChar <- 0
```

```
plot(NULL, xlim = c(0, 8), ylim = c(0, 5),
xaxt = "n", yaxt = "n", ylab = "", xlab = "",
main = "26 standard plotting characters")

while (plotChar < 26) {
  if (xCounter < 7) {
    xCounter <- xCounter + 1
  } else {
    xCounter <- 1
    yCounter <- yCounter + 1
  }
points(xCounter, yCounter, pch = plotChar, cex = 2)
text(xCounter, (yCounter - 0.3), plotChar)
plotChar <- plotChar + 1
}
```



#### Annotating Plots Plot accepts main title, subtitle, X label, Y label as standard arguments. plot(x, y, main="…", sub="…", xlab="…", ylab="…")

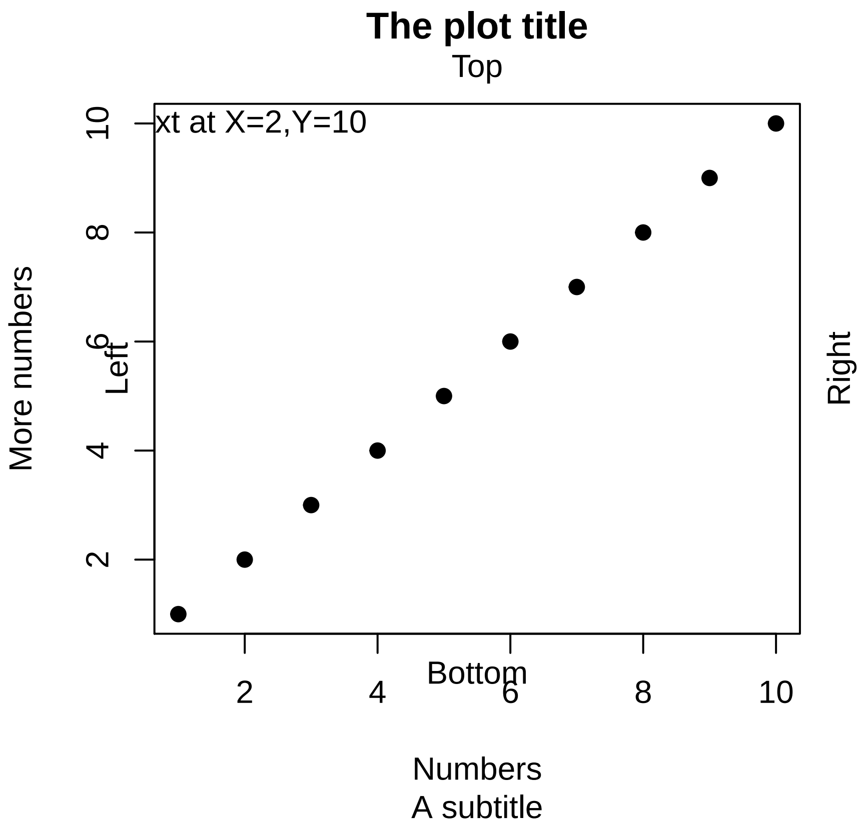
Write text directly into the margin of a plot. mtext(text="…", side=…)

Write text in the plot at x,y. text(x, y, labels="…")

Produce a legend for the plot. legend(x, y, legend=…)

Use locator() to find the x,y coordinates from mouse click within a plot.

```r
par(mfrow = c(1, 1))
par(bg = "white", fg = "black",cex = 1)
par(oma = c(1, 1, 1, 1))
par(mar = c(5, 4, 4, 2) + 0.1)

plot(1:10, main = "The plot title",
     sub ="A subtitle",
     xlab = "Numbers",
     ylab = "More numbers")

mtext(c("Bottom", "Left", "Top","Right"), c(1, 2, 3, 4), line=.5)
text(2, 10, "Text at X=2,Y=10")
```



Use of Colour in R

Colour is usually expressed as hexadecimal code of Red, Green, and Blue counterparts. R supports numerous colour palettes which are available through several "colour" functions.

```r
rainbow(10)
```

```
##  [1] "#FF0000FF" "#FF9900FF" "#CCFF00FF" "#33FF00FF" "#00FF66FF"
##  [6] "#00FFFFFF" "#0066FFFF" "#3300FFFF" "#CC00FFFF" "#FF0099FF"
```

```
heat.colors(10)
```

```
##  [1] "#FF0000FF" "#FF2400FF" "#FF4900FF" "#FF6D00FF" "#FF9200FF"
##  [6] "#FFB600FF" "#FFDB00FF" "#FFFF00FF" "#FFFF40FF" "#FFFFBFFF"
```

```
terrain.colors(10)
```

```
##  [1] "#00A600FF" "#2DB600FF" "#63C600FF" "#A0D600FF" "#E6E600FF"
##  [6] "#E8C32EFF" "#EBB25EFF" "#EDB48EFF" "#F0C9C0FF" "#F2F2F2FF"
```

```
topo.colors(10)
```

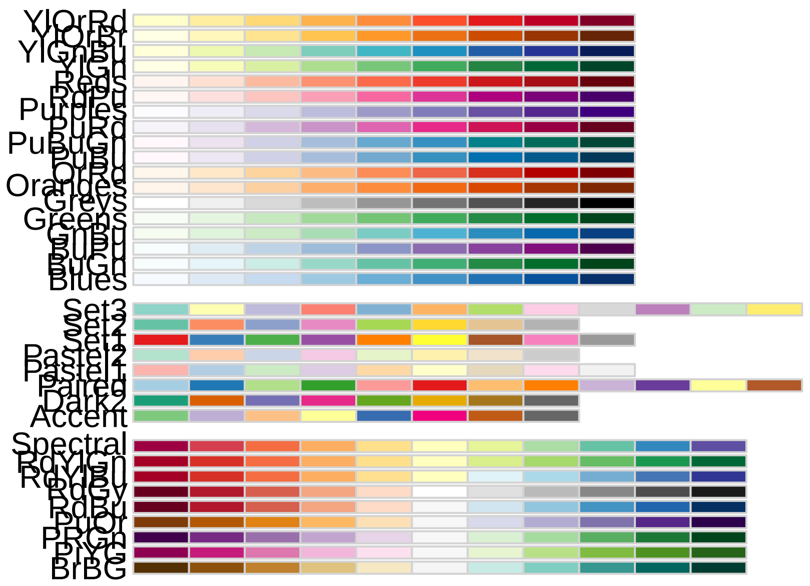
```
##  [1] "#4C00FFFF" "#0019FFFF" "#0080FFFF" "#00E5FFFF" "#00FF4DFF"
##  [6] "#4DFF00FF" "#E6FF00FF" "#FFFF00FF" "#FFDE59FF" "#FFE0B3FF"
```

```
cm.colors(10)
```

```
##  [1] "#80FFFFFF" "#99FFFFFF" "#B3FFFFFF" "#CCFFFFFF" "#E6FFFFFF"
##  [6] "#FFE6FFFF" "#FFCCFFFF" "#FFB3FFFF" "#FF99FFFF" "#FF80FFFF"
```

RColorBrewer is a add on package that provides a series of well defined colour palettes. The colours in these palettes are selected to permit maximum visual discrimination.

```
library("RColorBrewer")
display.brewer.all(n = NULL, type = "all", select = NULL, exact.n = TRUE)
```



Session information

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.3 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] RColorBrewer_1.1-2 dplyr_0.8.3        knitr_1.23
## [4] devtools_2.1.0     usethis_1.5.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.1        compiler_3.6.1    pillar_1.4.2
##  [4] prettyunits_1.0.2 remotes_2.1.0     tools_3.6.1
##  [7] zeallot_0.1.0     testthat_2.1.1    digest_0.6.20
## [10] pkgbuild_1.0.5    pkgload_1.0.2     evaluate_0.14
## [13] memoise_1.1.0     tibble_2.1.3      pkgconfig_2.0.2
## [16] rlang_0.4.0       cli_1.1.0         yaml_2.2.0
## [19] xfun_0.8          withr_2.1.2       stringr_1.4.0
## [22] vctrs_0.2.0       desc_1.2.0        fs_1.3.1
## [25] rprojroot_1.3-2   tidyselect_0.2.5  glue_1.3.1
## [28] R6_2.4.0          processx_3.4.1    fansi_0.4.0
## [31] rmarkdown_1.14    sessioninfo_1.1.1 purrr_0.3.2
## [34] callr_3.3.1       magrittr_1.5      backports_1.1.4
## [37] ps_1.3.0          htmltools_0.3.6   assertthat_0.2.1
## [40] utf8_1.1.4        stringi_1.4.3     crayon_1.3.4
```

The document was processed on 2019-09-11.