Contents

1. GitHub Tutorial

1.1 Getting Started With GitHub

What is Git?

Git is an online versioning tool that allows Data Science Dojo to continually push new content and issue patches to our course material. In the future, if Data Science Dojo offers new modules, alumni who are still subscribed to our repository may continue to receive these updates.

Setting up a GitHub Account

To set up an account, go to `https://github.com/join`.

Your GitHub Username

Go to your GitHub profile. Your username should be listed underneath your name and portrait. In the example below the GitHub username is "tempylionheart".

Installing Git Bash

The git clients for different operating systems can be downloaded at `git-scm.com`.

Cloning the DSD Bootcamp Repository

To clone the bootcamp repo over HTTPs, you must:

1. Open the Terminal (for Linux and Mac), or Git Bash(for Windows)
2. Navigate to the directory where you want to put this repo

```
cd [MyBootCampFolderFilePath]
```

3. Clone by typing

```
$ git clone https://github.com/datasciencedojo/
  bootcamp.git
```

4. Input the username and password of your own Github account (Make sure we already added you as a contributor of this repo) (A video tutorial of “clone” can be found on Youtube)

Synchronize the Newest Version of the Bootcamp Repository

Data Science Dojo regularly update the contents in the bootcamp repo. To synchronize your local bootcamp repo to the newest version:

1. Open the Terminal (for Linux and Mac), or Git Shell (for Windows)
2. Navigate to the directory of your local DSD bootcamp repo

```
cd [MyBootCampFolderFilePath]/bootcamp
```

3. Pull the changes

```
$ git pull origin master
```

4. Input the username and password of your own Github account

2. Introduction to R

2.1 Introducing R

2.1.1 What is R?

R is a language and environment for statistical computing and graphics. It provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology and R provides an Open Source route to participation in that activity. [1]

R is open source and highly extensibile. There are currently 6438 packages in the Comprehensive R Archive Network ("CRAN") package repository.

In this chapter we will go over R data types, basic operations, reading and writing data, statistical simulation, and basic plotting systems.

2.1.2 R is Vectorized

```
f = 1 * 11 + 2 * 22 + 3 * 33

> A = c(1, 2, 3)
> B = c(11, 22, 33)

> f = A * B
> f
[1] 11 44 99
```

The vectorization of R makes it good for statistics and data.

[1]The R Project for Statistical Computing: `http://www.r-project.org/`

2.1.3 R for Data Science

There are various advantages and disadvantages to using R for data science. One of its advantages is that it was designed for statistical analysis, making it more straightforward. For example, linear regression can be performed with a single line of R. Contrastingly in Python, this requires the use of several third-party libraries to represent the data (NumPy), perform the analysis (SciPy), and visualize the results (mat-plotlib).

One of the disadvantages of R is that it does not scale well with large data. Big companies like Google use R to play with data and experiment with new machine learning methods. If it works well and they decide to move forward, they use something like C. However, this doesn't mean that R can't be used for big data!

2.2 R Data Types

2.2.1 Hello World

```
C <- "Hello World" # type Enter
print(c) # print some text
# anything after a hash (#) is a comment
```

2.2.2 Basic Data Types

There are five basic classes (atomic classes) of R. They are character, numeric, integer, complex, and logical.

Among these classes the assignment operators are as follows:

```
n <- 10   # assign value 10 to variable 'n'
0.3 -> s  # the arrow can go both ways
m = TRUE   # can also use equal (=) operator for
   assignment
```

There are some differences between **<-** and **=** mainly involving the scope. To simplify this, Google's R style guide simplifies the issue by prohibiting the **=** as an assignment operator. You can read more about this at: `http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html#assignment`

2.2.3 Numbers

Integers in R are generally treated as numeric objects (e.g. double precision real numbers like 3.1415926...) If you explicitly want an integer, you need to specify using the L suffix. For example, entering 1 gives you a numeric object while entering 1L explicitly gives you an integer.

There is also a special number **Inf** which represents infinity (e.g. 1/0). **Inf** can be used in ordinary calculations (e.g. 1/ Inf is 0).

Keep in mind that, in general, all objects in R are case sensitive.

2.2.4 Dates and Times

Dates are represented by the date class. Time is represented by the POSIXct class, a very large integer. POSIXlt class is a list which stores a lot of useful metadata.

```
> x <- as.Date("2015-03-26")
> x
```

```
> x <- Sys.time()
> x
[1] "2015-03-23 21:28:10 PDT"
> p <- as.POSIXlt(x)
> names(unclass(p)) # unclass(p) is a list object
 [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"
    "wday"   "yday"
 [9] "isdst"  "zone"   "gmtoff"
> p$sec
[1] 10.89086
```

Utilize the **strptime** function in case your times are written in a different format as characters. For the **formatting strings**, check **?strptime** for details

```
> timeString <- "March 26, 2015 12:30"
> x <- strptime(timeString, "%B %d, %Y %H:%M")
> class(x)
[1] "POSIXlt" "POSIXt"
> x
[1] "2015-03-26 12:30:00 PDT"
```

2.2.5 Compound Objects

R has various compound objects. Some of these are vector, list, factor, matrix, and data frame.

2.2.6 Vector

The most basic of the compound objects is a vector. A vector can only contain objects of the same class.

Empty vectors can be created with the **vector()** function.

```
>  x  <-  vector("numeric",  length  =  10)
>  x
[1]  0  0  0  0  0  0  0  0  0  0
```

The **c()** function can be used to create vectors of objects.

```
> x  <-  c(0.5,  0.6) #number
> x  <-  c(TRUE,  FALSE) # logical
> x  <-  c(T,  F) #logical
> x  <-  c("a",  "b",  "c") #character
> x  <-  c(1+0i,  2+4i) #complex
```

The : operator is used to create integer sequences.

```
: operator
> x = 1:20
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
    19 20
```

2.2.7 List

List is a special type of vector. There are two characteristics of this type of vector.

1. It can contain elements of different classes (basic or compound).
2. Each element of a list can have a name.

Lists are a very important data type of R, so it is important to familiarize yourself with them.

```
> x  <-  list(1,  "a",  TRUE,  1  +  4i)
> x
[[1]]
[1] 1
[[2]]
[1] "a"
[[3]]
[1] TRUE
[[4]]
[1] 1+4i
```

```
> x <- list(a=c(T,T,F,F), b=2)
> x
$a
[1]  TRUE  TRUE FALSE FALSE

$b
[1] 2
```

2.2.8 Factor

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as a numerical vector, where each integer has a label. Using factors with labels is better than using integers. This is because factors are self-describing; having a variable with the values of "Male" and "Female" is better than a variable with values of 1 and 2.

```
> x  <-  factor(c("yes",  "yes",  "no",  "yes",  "no"))
> x
[1] yes yes no  yes no
Levels: no yes
```

```
> table(x)
x
 no yes
  2   3
```

The order of the levels in a **factor()** can be set using the levels argument. This can be important in linear modeling, as the first level is used as the baseline.

```
> x  <-  factor(c("yes",  "yes",  "no",  "yes",  "no"))
> x
[1] yes yes no  yes no
Levels: yes no
> x  <-  factor(x,levels=c("no","yes"))
> x
[1] yes yes no  yes no
Levels: no yes
```

2.2.9 Matrix

A matrix is a vector with a dimension attribute. The dimension attribute itself is an integer vector, with a length of 2 (nrow,ncol).

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
```

```
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

A matrix is constructed column-wise, meaning entries can be thought of as starting in the upper left corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m) = c(2,5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> n <- 1:10
> dim(n) = c(3,5)
Error in dim(n) = c(3, 5) :
  dims [product 15] do not match the length of object [10]
```

A matrix can be created by column-binding or row-binding with **cbind()** and **rbind()**.

```
> dim(m)
> 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
```

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m)
NULL
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
A 1 3
b 2 4
```

2.2.10 Data Frames

Data frames are used to store tabular data. Unlike matrices which must have each element belonging to the same class, data frames can store different classes of objects in each column. Data frames also have a special attribute called (row.names). Finally, data frames are usually created by calling **read.table()** or **read.csv()**

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F)
  )
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

2.2.11 Coercion

When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.

```
> y <- c(1.7, "a") #character
> y
[1] "1.7" "a"
>  y <- c(TRUE, 2) #numeric
> y
[1] 1 2
```

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

```
> x <- 0:6
> class(x)
[1] "integer"

> as.numeric(x)
[1] 0 1 2 3 4 5 6
```

```
> as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"

> as.complex()
complex(0)

> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

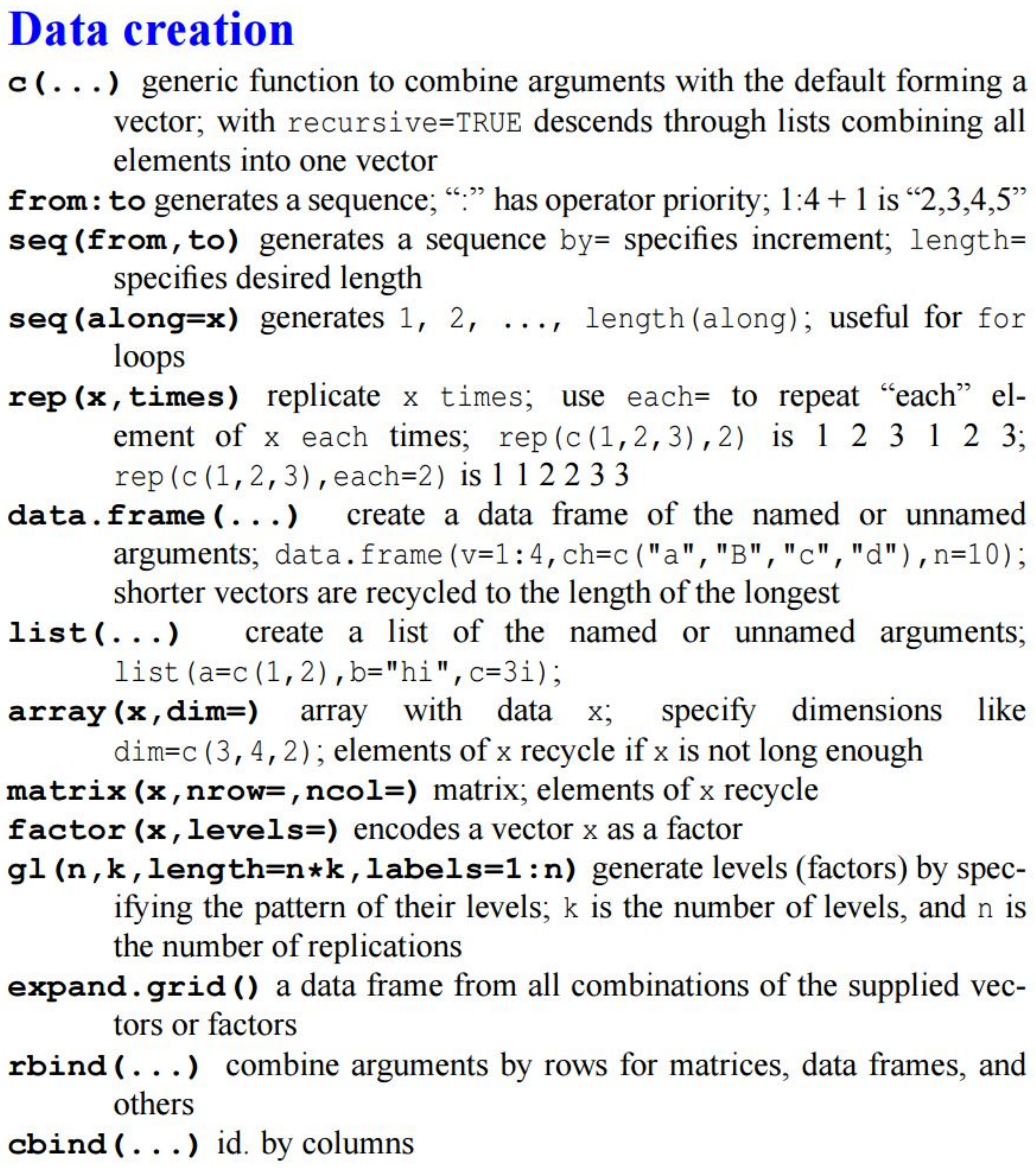
```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> x
[1] "a" "b" "c"
> as.logical(x)
[1] NA NA NA
```

2.2.12 Missing Values

Missing values are denoted by **NA** or **NaN** for undefined mathematical operations. **NA** means a missing value and has various forms (e.g. NA_integer, NA_character, etc.). **NaN** means the value is not a number. **NaN** is also **NA** but the converse of this is not true.

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE

> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```



Data creation

`c(...)` generic function to combine arguments with the default forming a vector; with `recursive=TRUE` descends through lists combining all elements into one vector

`from:to` generates a sequence; ":" has operator priority; 1:4 + 1 is "2,3,4,5"

`seq(from,to)` generates a sequence `by=` specifies increment; `length=` specifies desired length

`seq(along=x)` generates `1, 2, ..., length(along)`; useful for `for` loops

`rep(x,times)` replicate `x` `times`; use `each=` to repeat "each" element of `x` `each` times; `rep(c(1,2,3),2)` is 1 2 3 1 2 3; `rep(c(1,2,3),each=2)` is 1 1 2 2 3 3

`data.frame(...)` create a data frame of the named or unnamed arguments; `data.frame(v=1:4,ch=c("a","B","c","d"),n=10)`; shorter vectors are recycled to the length of the longest

`list(...)` create a list of the named or unnamed arguments; `list(a=c(1,2),b="hi",c=3i)`;

`array(x,dim=)` array with data `x`; specify dimensions like `dim=c(3,4,2)`; elements of `x` recycle if `x` is not long enough

`matrix(x,nrow=,ncol=)` matrix; elements of `x` recycle

`factor(x,levels=)` encodes a vector `x` as a factor

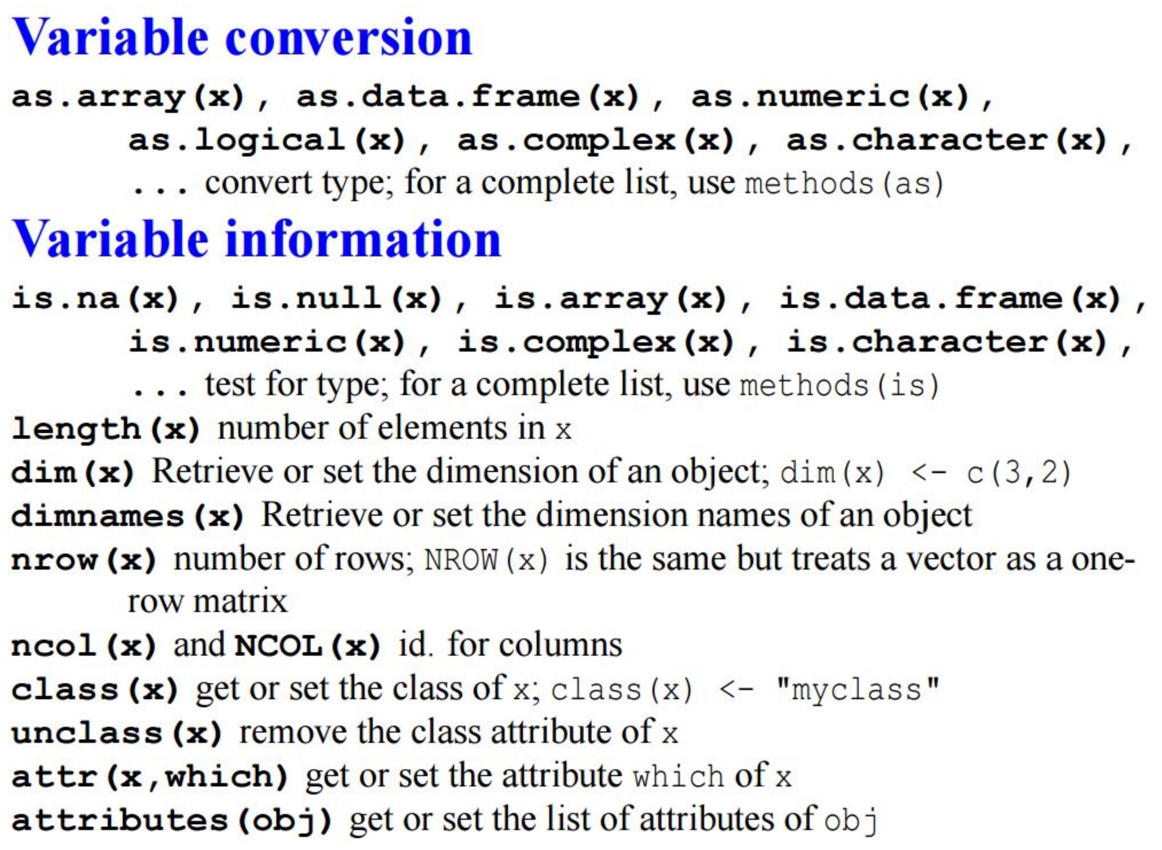
`gl(n,k,length=n*k,labels=1:n)` generate levels (factors) by specifying the pattern of their levels; `k` is the number of levels, and `n` is the number of replications

`expand.grid()` a data frame from all combinations of the supplied vectors or factors

`rbind(...)` combine arguments by rows for matrices, data frames, and others

`cbind(...)` id. by columns

Figure 2.1: Snippet from the R cheatsheet. Source: `http://cran.r-project.org/doc/contrib/Short-refcard.pdf`



Variable conversion

as.array(x), as.data.frame(x), as.numeric(x), as.logical(x), as.complex(x), as.character(x), ... convert type; for a complete list, use `methods(as)`

Variable information

is.na(x), is.null(x), is.array(x), is.data.frame(x), is.numeric(x), is.complex(x), is.character(x), ... test for type; for a complete list, use `methods(is)`

length(x) number of elements in `x`

dim(x) Retrieve or set the dimension of an object; `dim(x) <- c(3,2)`

dimnames(x) Retrieve or set the dimension names of an object

nrow(x) number of rows; `NROW(x)` is the same but treats a vector as a one-row matrix

ncol(x) and **NCOL(x)** id. for columns

class(x) get or set the class of `x`; `class(x) <- "myclass"`

unclass(x) remove the class attribute of `x`

attr(x,which) get or set the attribute `which` of `x`

attributes(obj) get or set the list of attributes of `obj`

Figure 2.2: Snippet from the R cheatsheet. Source: `http://cran.r-project.org/doc/contrib/Short-refcard.pdf`

2.2.13 Exercise

To sum up what we learned in this section. R has basic data types and compound objects. It is also important to keep in mind the attributes of coercion and missing values in R.

In the following exercise we will declare two vectors with three elements each. Call these two vectors **x** and **y** and do the following:

1. Use **rbind()** and **cbind()** to create a matrix from these two variables. Observe the difference in structure resulting from these two functions.
2. Name the columns of the matrix.
3. Create a data frame from **x** and **y** and give the columns appropriate names.

```
> x <- c(1, 2, 3)
> y <- c(4, 5, 6)
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y    4    5    6
```

```
> cbind(x, y)
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
> a = cbind(x, y)
> colnames(a) <- c("x", "y")
```

2.3 Basic Operations

In this section we will be covering subsetting, control structures, build-in functions, user written functions, apply, packages, and the working directory and R script.

2.3.1 Subsetting

There are a number of operators that can be used to extract subsets of R objects. For example, **[** always returns an object of the same class as the original object. With one exception, it can be used to select more than one element. **[[** is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame. **$** is used to extract elements of a list or data frame by name. The semantics of **$** are similar to **[[**.

```
> x  <-  c("a",  "b",  "c",  "c",  "d",  "a")
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
> x > "a"
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> x[x>"a"]
[1] "b" "c" "c" "d"
```

```
> x  <-  matrix(1:6, 2,  3)
> x[1,2]
[1] 3
> x[1,] # Entire first row.
[1] 1 3 5
> x[,2] # Entire second column.
[1] 3 4
```

By default, when a single element of a matrix is retrieved, it is returned as a vector with a length of 1 rather than a 1 x 1 matrix. This behavior can be turned off by setting drop equal to FALSE.

```
> x  <-  matrix(1:6,  2,  3)
> x[1,2]
[1] 3
> x[1,2,drop=FALSE]
     [,1]
[1,]    3
> x[1,]
[1] 1 3 5
```

Similarly, subsetting a single column or a single row will give you a vector, not a matrix by default.

```
> x[1,,drop=FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

```
> x  <-  list(foo  =  1:4,  bar  =  0.6)
> x[1]
$foo
[1] 1 2 3 4
> x$foo
[1] 1 2 3 4
> x$bar
[1] 0.6
> x["bar"]
$bar
[1] 0.6
> x[["bar"]]
[1] 0.6
```

Below is an example of extracting multiple elements of a list:

```
>  x <- list(foo  =  1:4,  bar  =  0.6,  baz  =  "hello")
>  x[c(1, 3)]
$foo
[1]  1  2  3  4

$baz
[1]  "hello"
```

Partial matching of names is allowed with **[[** and **$**

```
> x <- list(addedName = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a",exact=FALSE]]
[1] 1 2 3 4 5
```

2.3.2 Control Structures

Conditional:

```
if (logical.expression) {
   statements
}
else if (another.logical.expression) {
   statements
}
else {
   alternative.statements
}
else if, else branch is optional
```

for-loop:

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

repeat-loop:

```
i=1
repeat {
 i= i+1
 if (i>=10) break # the only
# way to get out of the loop
}
```

while-loop:

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

R can support looping however, keep in mind:

1. Looping is not recommended for compound objects like lists, vectors, or data.frames, etc. For example, a loop could be used to multiply all elements by 2, but the loop is implicit.

```
> x <- c(1, 2, 3)
> x * 2
[1] 2 4 6
```

2. Control structures mentioned here are primarily useful for writing programs. For command-line interactive work, the ***apply** functions (lapply, sapply, etc.) are more useful.

2.3.3 Built-in Functions

Numeric Functions

```
abs(x)  absolute value
sqrt(x) square root
ceiling(x)  ceiling(3.475) is 4
floor(x)  floor(3.475) is 3
trunc(x)  trunc(5.99) is 5
round(x, digits=n)  round(3.475, digits=2) is 3.48
...
```

Character Functions

```
paste(..., sep="")  Concatenate strings after using sep
   string to seperate them.
paste("x",1:3,sep="") returns c("x1","x2" "x3")
paste("x",1:3,sep="M") returns c("xM1","xM2" "xM3")

grep(pattern, x , ignore.case=FALSE, fixed=FALSE)

Search for pattern in x. If fixed =FALSE then pattern is a
    regular expression. If fixed=TRUE then pattern is a
   text string. Returns matching indices.
grep("A", c("b","A","c"), fixed=TRUE) returns 2
...
```

Statistical Functions

```
rnorm(), dunif(), mean(), sum()
```

You can find a more complete list of other built-in functions at: `http://www.statmethods.net/management/functions.html`

2.3.4 User-written Functions

```
foo <- function(x,y=1,...) {
  cat("extra args:",...,"\n")
  sqrt(x)+sin(y)
}
foo(1,2,3,"bar")
foo(1)
foo(y=3,x=7) # named arguments in any order
```

Type the function name to see the code. This works for any function. foo

Tip The three dots used in the code allow for an arbitrary number and variety of arguments. They also allow for the passing of arguments on to other functions.

2.3.5 str Function

str compactly displays the internal structure of an R object (data or function). It is a diagnostic function that acts as an alternative to **summary()** for a data object. **str** is especially well suited to compactly display the abbreviated contents of a possibly nested list. It also tells you what is in the object and the arguments of the function.

Try out: **str(lm)**

2.3.6 apply

lapply loops over a list and evaluates a function for each element. **sapply** is the same as **lapply** and tries to simplify the result.

```
> lapply(c( -1, -5), abs)
[[1]]
[1] 1

[[2]]
[1] 5
```

```
> sapply(c( -1, -5), abs)
[1] 1 5
```

apply applies a function over the margins of an array.

```
> x <- matrix(c(1, 2, 3, 4), 2, 2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> apply(x, 1, sum)
[1] 4 6
> apply(x, 2, sum)
[1] 3 7
```

tapply applies a function over the subsets of a vector.

```
> score <- c(90, 79, 94, 85)
> gender <- factor(c("Male", "Female", "Female", "Male"))
> tapply(score, gender, mean)
Female   Male
  86.5   87.5
```

split is an auxiliary function. A common idiom is split followed by lapply

```
> score <- c(90, 79, 94, 85)
> gender <- factor(c("Male", "Female", "Female", "Male"))
> splitted <- split(score, gender)
> splitted
$Female
[1] 79 94
$Male
[1] 90 85
> lapply(splitted, mean)
$Female
[1] 86.5
$Male
[1] 87.5
```

mapply is a multi-variable input. It applies a function in parallel over a set of arguments.

```
> mapply(rep, c(0,2), c(3,5)) # input certain combinations
## <- list(rep(0,3), rep(2,5)), it vectorizes a function
[[1]]
[1] 0 0 0

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

Don't worry if you forget the logics of **apply** functions. You can always use **lapply**, **str(lapply)**, or Google/Bing to check.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

2.3.7 Packages

For almost everything you want to do with R, there is probably a package written to do just that. A list of packages can be found in the official repository CRAN: `http://cran.fhcrc.org/web/packages/`. If you need a package, it can be installed very easily from within R using the command:

```
install.packages("packagename") # if package already
   installed, it'll bypass
```

Libraries in Github can be installed using the devtools library.

2.3.8 Working directory and R script

You can set the working directory from the menu if using the R-gui (Change dir…) or from the R command line:

```
setwd("C:\\MyWorkingDirectory")
setwd("C:/MyWorkingDirectory") # can use forward slash
setwd(choose.dir()) # opens a file browser

getwd()  # returns a string with
         # the current working directory
```

To see a list of the files in the current directory:

```
dir() # returns a list of strings of file names
dir(pattern=".R$") # list of files ending in ".R"
dir("C:\\Users") # show files in directory C:\Users
```

Run a script:

```
source("helloworld.R") # execute a script
```

2.4 Reading and Writing Data

In this section we will be covering reading and writing local flat files, reading and writing local Excel files, connection interfaces, reading XML/HTML files, reading and writing to JSON, connecting to a database, and the textual format.

2.4.1 Reading and Writing Local Flat Files

You can read local flat files as **read.table**, **read.csv**, **readLines**. CSV stands for "Comma Separated Values."

For this lab we will be working with the Titanic dataset. To procure this dataset, please visit the following link:
`https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv`
Right click and select "save as" to save it to your local directory. The code samples in the remainder of this chapter assumes that you have set your working directory to the location of the titanic.csv file.

```
> titanic_data <- read.csv("titanic.csv")
> head(titanic_data, 3)
  X Class  Sex   Age Survived Freq
1 1   1st Male Child       No    0
2 2   2nd Male Child       No    0
3 3   3rd Male Child       No   35
```

```
> line1 <- readLines("titanic.csv", 1)
> line1
[1] "\"\",\"Class\",\"Sex\",\"Age\",\"Survived\",\"Freq\""
```

You can write local flat files as **write.table**, **write.csv**, **writeLines**.

```
> write.table(titanic_data, "new_Titanic.csv")
```

Before reading or writing files, make sure to take note of the parameters. For example the **read.table** function is one of the most commonly used functions for reading data. It has a few important arguments to take into account:

- **file** is the name of a file, or a connection.
- **header** a logical value indicating the variables of the first line.
- **sep** is a string indicating how the columns are separated.
- **colClasses** is a character vector indicating the class of each column in the dataset.
- **nrows** are the number of rows in the dataset
- **comment.char** is character string indicating the comment character .
- **skip** is the number of lines to skip from the beginning.
- **stringsAsFactors** determines whether character variables be coded as factors.

2.4.2 Reading and Writing Excel Files

read.xlsx, write.xlsx, or read.xlsx2, write.xlsx2 (faster, but unstable)

```
# Install the xlsx library
> install.packages('xlsx')
# Load the library
> library(xlsx)
# Now you can Read the Excel file
> titanic_data <- read.xlsx("titanic3.xls", sheetIndex=1)
```

2.4.3 Connection Interfaces

In practice, we don't need to deal with the connection interface directly. Connections can be made to files or to other channels:

- **file** opens a connection to a file
- **gzfile** opens a connection to a file compressed with gzip
- **bzfile** opens a connection to a file compressed with bzip2
- **url** opens a connection to a webpage

Here is a simple example:

```
> con <- file("Titanic.csv", "r")
> titanic_data <- read.csv(con)
> close(con)

> con <- url("http://vincentarelbundock.github.io/
  Rdatasets/csv/datasets/Titanic.csv")
> another_data <- read.csv(con)
```

2.4.4 Reading XML/HTML Files

```
> library(XML) # you need to install this library
> url <- "http://www.w3schools.com/xml/simple.xml"
> doc <- xmlTreeParse(url, useInternal=TRUE) # also works
   for html
> rootNode <- xmlRoot(doc)
> rootNode[[1]]
<food>
  <name>Belgian Waffles</name>
  <price>$5.95</price>
  <description>Two of our famous Belgian Waffles with
     plenty of real maple syrup</description>
  <calories>650</calories>
</food>
```

```
> rootNode[[1]][[1]]
<name>Belgian Waffles</name>
> xpathSApply(rootNode, "//name", xmlValue)
[1] "Belgian Waffles" "Strawberry Belgian Waffles" "Berry-
   Berry Belgian Waffles" "French Toast" "Homestyle
   Breakfast"
```

- **xpath**: **/node**: top level node; **//node**: node at any level;
- **node[@attr-name]**: node with an attribute name;
- **node[@attr-name = ́bob']**: node with an attribute name = ́bob'

2.4.5 Reading and Writing JSON

JSON or JavaScript Object Notation or "JSON" is a common format for data from application programming interfaces ("APIs"). It is an alternative to XML.

```
> library(jsonlite)
> jsonData <- fromJSON("http://citibikenyc.com/stations/
   json")
> names(jsonData)
[1] "executionTime"   "stationBeanList"
> jsonData$stationBeanList[1,1:3]
  id       stationName availableDocks
1 72 W 52 St & 11 Ave             31
```

```
> head(iris, 3)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
    Species
1          5.1         3.5          1.4         0.2
   setosa
2          4.9         3.0          1.4         0.2
   setosa
3          4.7         3.2          1.3         0.2
   setosa

> iris2 <- toJSON(iris, pretty=TRUE)
```

2.4.6 Connect to a Database

JSONPackages: RmySQL, RpostresSQL, RODBC, RMONGO

```
> library(RMySQL)# load the library
> ucscDb <- dbConnect(MySQL(), user="genome", host="genome
   -mysql.cse.ucsc.edu")
> data <- dbGetQuery(ucscDb, "show databases;") # get the
   output of SQL query as data frame in R
> head(data)
            Database
1 information_schema
2            ailMel1
3            allMis1
4            anoCar1
5            anoCar2
6            anoGam1
> dbDisconnect(ucscDb) # don't forget to close the
   connection
```

2.5 Statistical Simulation

Probability distribution functions usualy have four functions associated with them. The functions are prefixed with:

- **d** for density, like dnorm()
- **r** for random number generation, like rnorm()
- **p** p for cumulative distribution, like pnorm()
- **q** for quantile function, like qnorm()

Below is an example of normal distribution:

```
> dnorm(0, mean = 0, sd = 1)
[1] 0.3989423
> dnorm(10, mean = 0, sd = 1)
[1] 7.694599e-23

> pnorm(0, mean = 0, sd = 1)
[1] 0.5
> pnorm(1, mean = 0, sd = 1)
[1] 0.8413447
> pnorm(100, mean = 0, sd = 1)
[1] 1
```

```
> qnorm(0.5, mean = 0, sd = 1)
[1] 0
> qnorm(0, mean = 0, sd = 1)
[1] -Inf
> qnorm(0.2, mean = 0, sd = 1)
[1] -0.8416212

> rnorm(4, mean = 0, sd = 1)
[1] -0.80938783 -0.07203091  0.99059330  0.69783570
```

Built-in standard distributions: Normal (**norm**), Poisson (**pois**), Binomial (**binom**), Exponential (**exp**), Gamma (**gamma**), Uniform (**unif**)

Reproducible random number: **set set.seed()** in advance

```
> rnorm(3, mean = 0, sd = 1)
[1] -1.1991719 -0.3227133  0.4802463
> set.seed(1)
> rnorm(3, mean = 0, sd = 1)
[1] -0.6264538  0.1836433 -0.8356286
> set.seed(1)
> rnorm(3, mean = 0, sd = 1)
[1] -0.6264538  0.1836433 -0.8356286
```

2.6 Plotting Systems (optional)

In this chapter we will be covering a few of the different kinds of plotting systems.

- **Base graphics** are constructed piecemeal. This system is conceptually simpler and allows plotting to mirror the thought process.
- **Lattice graphics** are entire plots created through a simple function call.
- **ggplot2 graphics** are an implementation of the Grammar of Graphics by Leland Wikinson. They combine concepts from both base and lattice graphics. To use this system you must install the ggplot2 library.

Fancier and more telling ones will be elaborated on further during the bootcamp! In the meantime, check out a list of different interactive visualizations in R: `http://ouzor.github.io/blog/2014/11/21/interactive-visualizations.html`

2.6.1 Basic Plotting Systems

```
> library(datasets)
> names(airquality)
[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day
  "
> plot(x = airquality$Temp, y = airquality$Ozone)
```

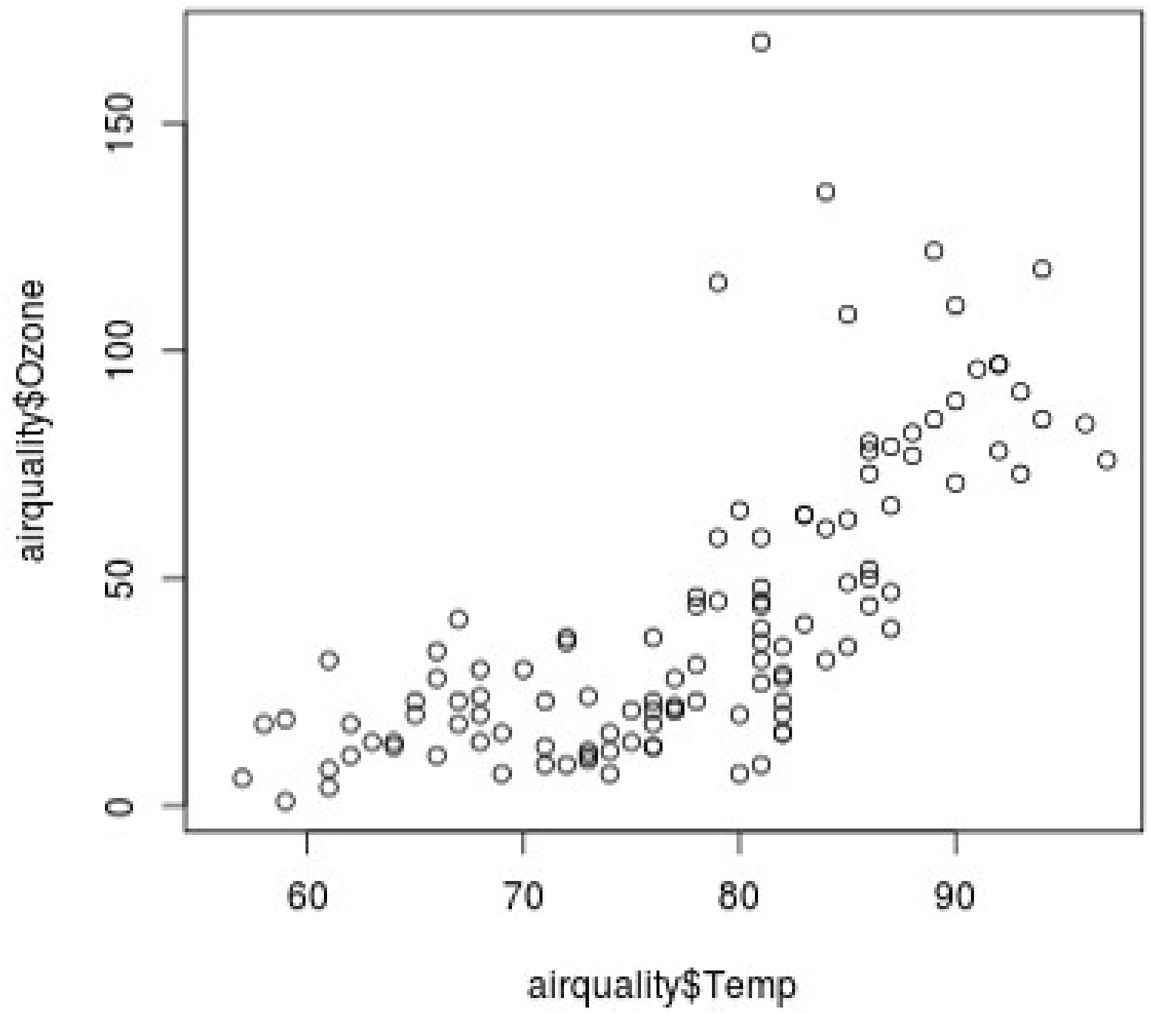


Figure 2.3: Scatter plot of temperature and ozone level

```
# par() function is used to specify global graphics
  parameters
# that affect all plots in an R session. Type ?par to see
  all
# parameters
> par(mfrow = c(1, 2), mar = c(4, 4, 2, 1), oma = c(0, 0,
  2, 0))
> with(airquality, {
+ plot(Wind, Ozone, main="Ozone and Wind")
+ plot(Temp, Ozone, main="Ozone and Temperature")
+ mtext("Ozone and Weather in New York City", outer=TRUE)
+ })
```

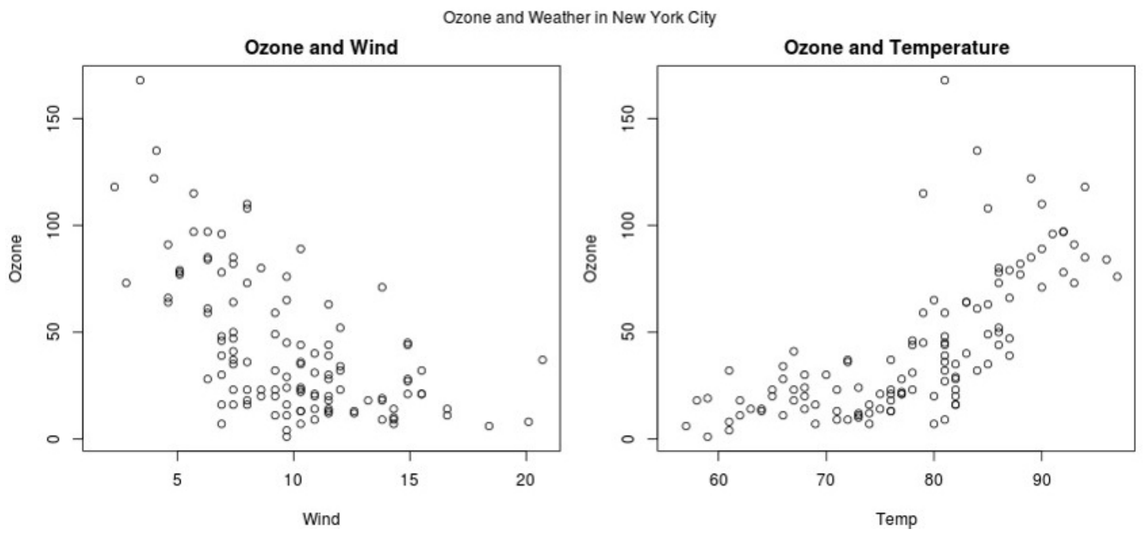


Figure 2.4: Correlation between ozone and weather conditions in New York City

There are various plotting functions that can be utilized.

- **lines** add lines to a plot, given a vector of x values and corresponding vector of y values.
- **points** adds a point to the plot.
- **text** adds text labels to a plot using specified x,y coordinates.
- **title** adds annotations to x,y axis labels, title, subtitles, and the outer margin.
- **mtext** adds arbitrary text to the inner or outer margins of the plot.
- **axis** specifies the axis ticks

2.6.2 Lattice Plotting Systems

```r
> library(lattice) # need to load the lattice library
> set.seed(10) # set the seed so our plots are the same
> x <- rnorm(100)
> f <- rep(1:4, each = 25) # first 25 elements are 1,
   second 25 elements are 2, ...
> y <- x + f - f * x+ rnorm(100, sd = 0.5)
> f <- factor(f, labels = c("Group 1", "Group 2", "Group 3
   ", "Group 4"))
> xyplot(y ~ x | f)
```

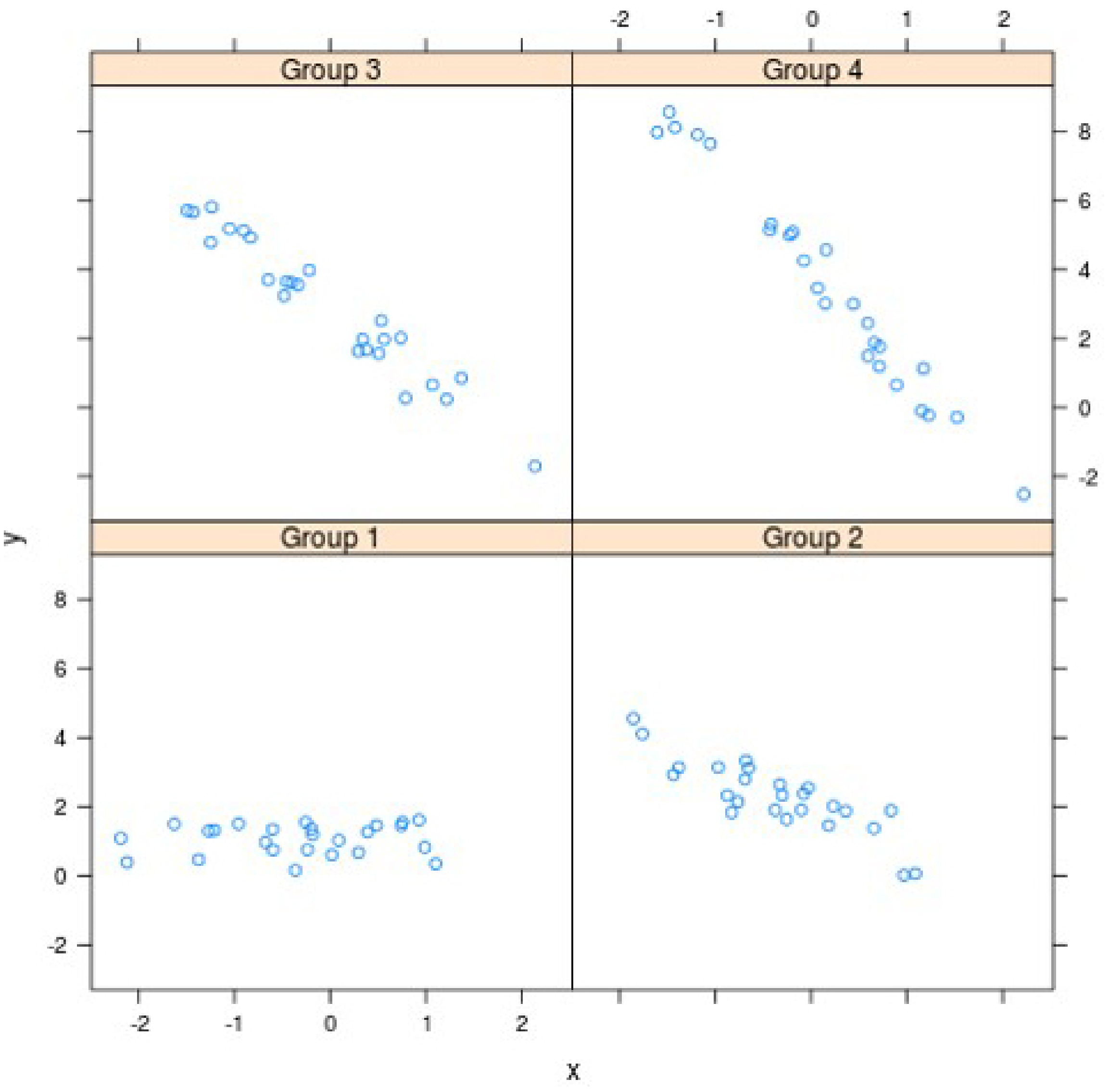


Figure 2.5: Example of simulated data by factor level

What more can you do with the plots?

```
> xyplot(y ~ x | f, panel = function(x, y, ...) {
    # call the default panel function for xyplot
    panel.xyplot(x, y, ...)
    # adds a horizontal line at the median
    panel.abline(h = median(y), lty = 2)
    # overlays a simple linear regression line
    panel.lmline(x, y, col = 2)
})
```

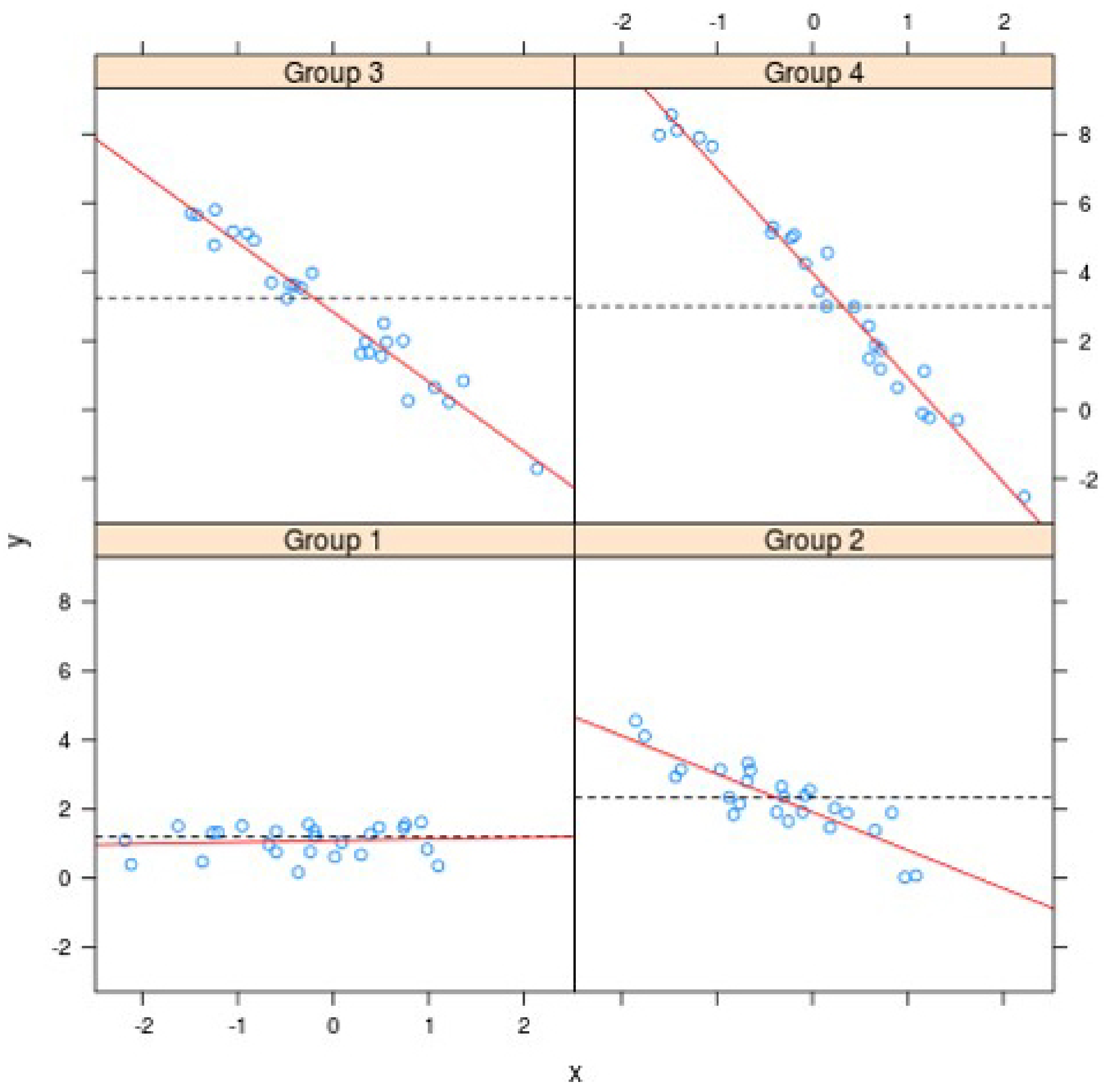


Figure 2.6: Adding linear trend lines to scatter plot

Plotting Functions:

xyplot() is the main function for creating scatterplots **bwplot()** is a box and whiskers plot or box plot **histogram()** is a histogram **stripplot()** is a box plot with actual points **dotplot()** is a plot with dots on "violin strings" **splom()** is scatterplot matrix, similar to **pairs()** in the base plotting system **levelplot()/contourplot()** plots image data

2.6.3 ggplot2 Plotting systems

Before using ggplots, you must install the ggplot2 library. Ggplots mix the elements of base and latice systems. A good tutorial on the three basic plotting systems can be found at: `https://sux13.github.io/DataScienceSpCourseNotes/4_EXDATA/Exploratory_Data_Analysis_Course_Notes.html`.

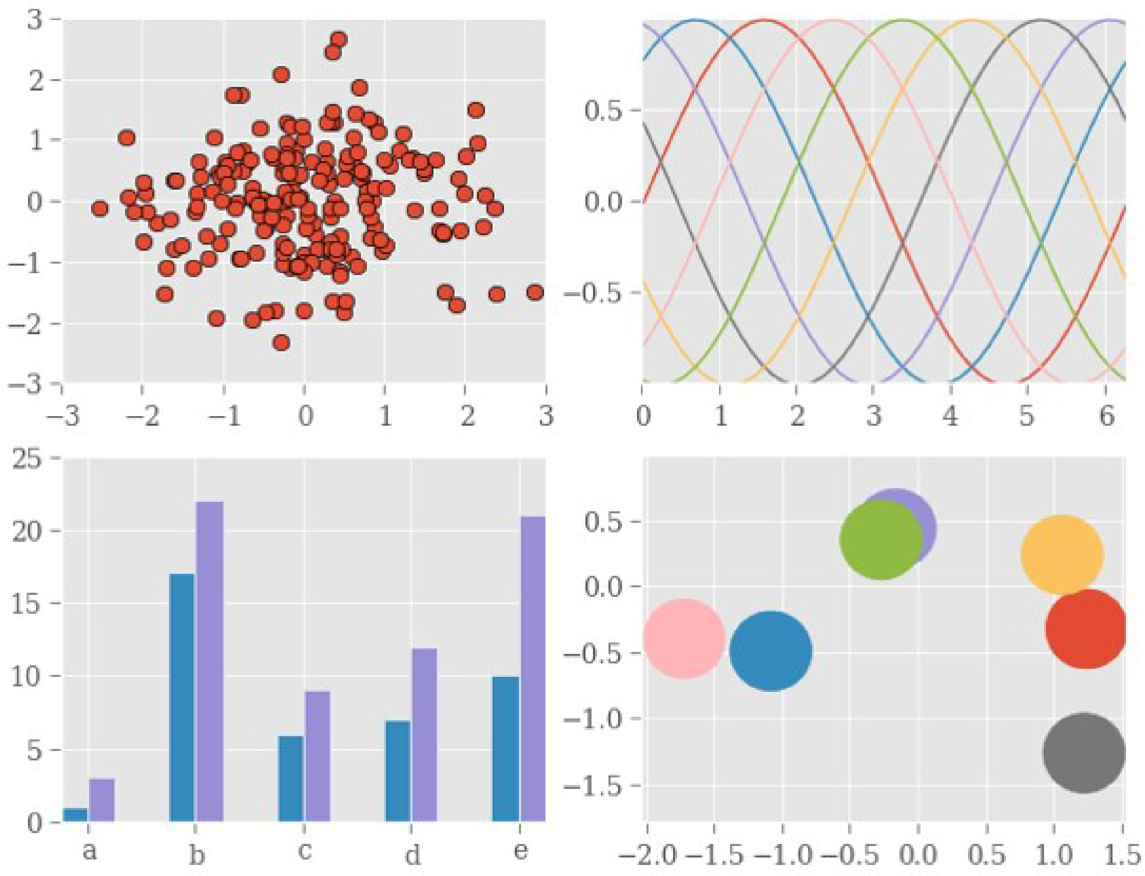


Figure 2.7: Example demonstrations of ggplots. Source: `http://matplotlib.org/examples/style_sheets/plot_ggplot.html`