

R Tutorial

- Introduce R

Yuhui Zhang
yuhui@datasciencedojo.com

Raja Iqbal
raja@datasciencedojo.com
datasciencedojo
unleash the data scientist in you

What is R?

R is a language and environment for statistical computing and graphics....R 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.

—*The R Project for Statistical Computing*, <http://www.r-project.org/>

Open source + highly extensibility + era of data science

→ Currently **6438 packages in CRAN** package repository (the official one)

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
```

Good for statistics
Good for data

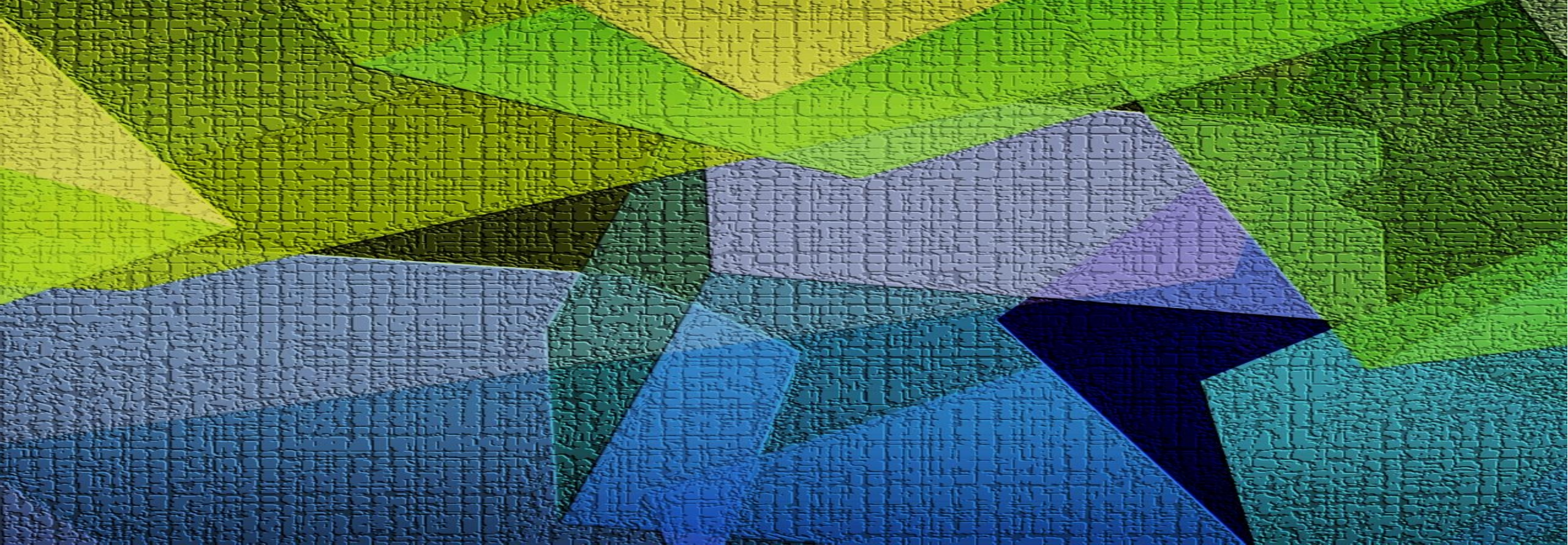
R for data science

Advantage: designed for statistical analysis, more straightforward. For example: linear regression can be performed with a single line in R. While in Python, it requires the use of several third-party libraries to represent the data (NumPy), perform the analysis (SciPy), and visualize the results (matplotlib).

Disadvantage: does not scale well with large data. Big companies like Google use R as their "data sandbox" to play with data and experiment with new machine learning methods. If it works well, further use something like C. This doesn't mean we cannot use R for big data – sample!

Overview

- R data types
- Basic operations
- Reading and writing data
- Statistical simulation
- Basic plotting systems (overview)



R Tutorial

- R Data Types

Yuhui Zhang
yuhui@datasciencedojo.com

Raja Iqbal
raja@datasciencedojo.com
datasciencedojo
unleash the data scientist in you

Hello world

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

Basic data types

Five basic classes (atomic classes)

Character, Numeric, Integer, Complex, Logical

Assignment operator

```
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
```

<- and = do have **some difference**, mainly about the scope...
Google's R style guide simplifies this issue by prohibiting the
"=" for assignment. Not a bad choice. (<http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html#assignment>)

Numbers

Integer:

Numbers in R are generally treated as numeric objects (i.e. double precision real numbers like 3.1415926...)

If you explicitly want an **integer**, you need to specify the L suffix

Ex: Entering 1 gives you a numeric object; entering 1L explicitly gives you an integer.

Infinity (Inf):

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.

Case sensitive! And in general, objects in R is case sensitive.

Dates and times

Date: represented by the date class

Time: POSIXct class (a very large integer);
POSIXlt class (as a list, stores a bunch of useful meta-data)

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

Dates and times

```
> 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"     "yday"     "yday"
[9] "isdst"    "zone"     "gmtoff"
> p$sec
[1] 10.89086
```


Dates and times

strptime function in case your times are written in a different format as characters

```
> 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"
```

For the **formatting strings**, check **?strptime** for details

Compound objects

Vector,
List,
Factor,
Matrix,
Data frame, etc.

Vector

The most basic object is a vector

A vector can only contain objects of the same class

Vector - using the vector() function

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
```

Vector - creating vectors (1)

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
```

Vector - creating vectors (2)

: 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
```

The : operator is used to create integer sequences.

List

List is a special type of vector:

1. Can contain elements of different classes (either basic class or compound class);
2. Each element of list can have name.

Lists are a very important data type in R and you should get to know them well.

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

List

```
> x <- list(a=c(T,T,F,F), b=2)
```

```
> x
```

```
$a
```

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

```
$b
```

```
[1] 2
```

Factor

Factors are used to represent categorical data.

Factors can be unordered or ordered.

One can think of a factor as an integer vector where each integer has a label.

Using factors with labels is better than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has values 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
```

Factor - changing the order of levels

The **order** of the levels can be set using the `levels` argument to `factor()`.

This can be important in **linear modeling** because the first level is used as the baseline level.

```
> 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
```


Matrix

Matrix is vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 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
```

Matrix - column-wise construction

Matrix is constructed *column-wise*, so entries can be thought of 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]
```

Matrix - cbind() and rbind()

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

```
> 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
```

Matrix - naming matrix

```
> 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
```


Data frames!

Data frames are used to store tabular data.

Unlike matrices, data frames can store different classes of objects in each column; while matrices must have every element be the same class.

Data frames also have a special attribute called `row.names`

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
```

Coercion

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

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

Coercion - explicit coercion

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

Missing values

Missing values are denoted by NA or NaN for undefined mathematical operations

NaN: Not a Number (like $2/0$)

NA: a missing value and has various forms - NA_integer_, NA_character_, etc.

NaN value is also NA but the converse 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

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`

Summary of R Data Types

1. Basic data types (dates and times)
2. Compound objects
3. Coercion
4. Missing values

Exercise

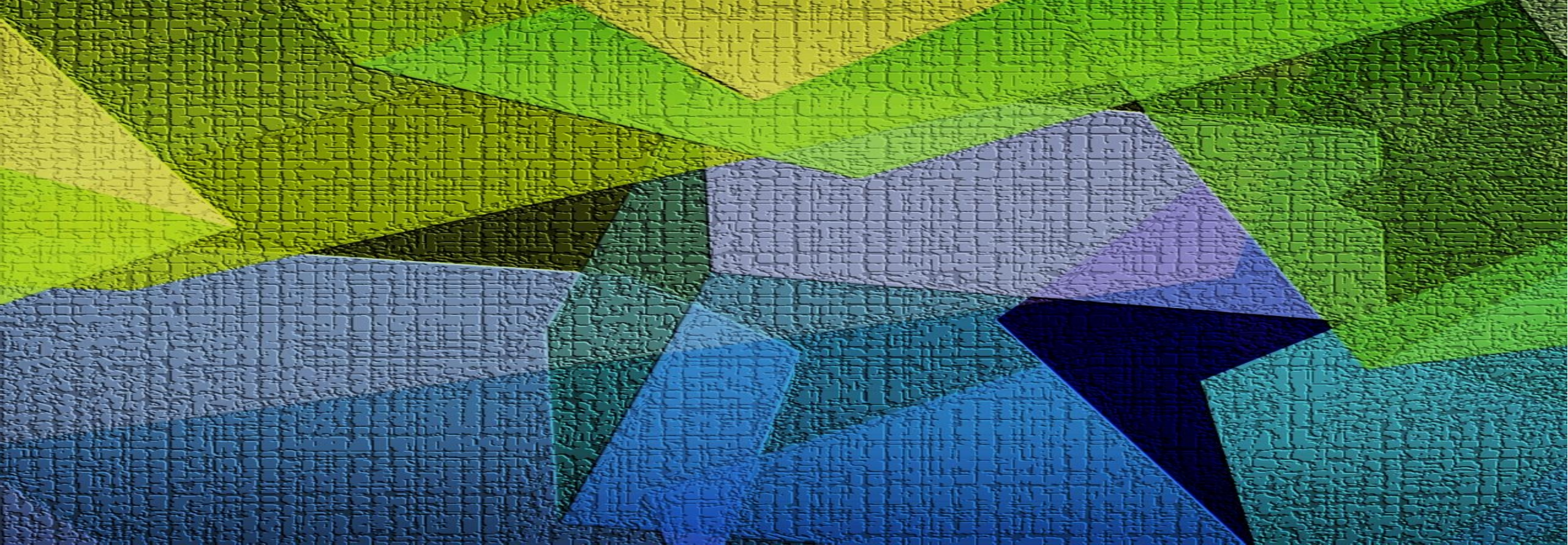
Declare two vectors with 3 elements each. Call these *x* and *y*. Now do the following:

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

Exercise

```
| > 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")
```

R Tutorial

- Basic Operations

Yuhui Zhang
yuhui@datasciencedojo.com

Raja Iqbal
raja@datasciencedojo.com
datasciencedojo
unleash the data scientist in you

Subsetting

There are **a number of** operators that can be used to extract subsets of R objects:

[always returns an object of the same class as the original object. Can be used to select more than one element (there is one exception).

[[is used to extract elements of a list or a data frame
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.
Semantics are similar to that of **[[**.

Subsetting – in vector and matrix

```
> 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
```


Subsetting - in matrix (the exception of [])

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. This behavior can be turned off by setting `drop = 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
```

Subsetting - in list

```
> 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
```

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"
```

Subsetting - partial matching

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
```

Control structures: conditional

Conditional:

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

Control structures: loop

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)  
}
```

Control structures: loop

R can support looping. **But**

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 (like lapply, sapply, etc.) are more useful.

Build-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 separate 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
...

Build-in functions

Statistical functions

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

More complete list:

<http://www.statmethods.net/management/functions.html>

User-written function

```
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
```

The three-dots allows:

1. an arbitrary number and variety of arguments
2. passing arguments on to other functions

Type function name to see code (works for any function)

```
foo
```

str function

`str` compactly displays the internal structure of an R object (data object or function)

1. A diagnostic function and an alternative to `summary()` for data object

2. It's especially well suited to compactly display the (abbreviated) contents of (possibly nested) list

It tells you what's in the object.

Used for functions: tells you the function's arguments.

Try: `> str(lm)`

*apply

lapply: loop over a list and evaluate a function on each element

sapply: same as lapply and try to simplify the result

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

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] 5
```

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

```
[1] 1 5
```

*apply

apply: apply 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
```

*apply

tapply: apply a function over 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
```


*apply

split: 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
```

*apply

mapply: multi-variable input, 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
```

*apply

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)
```

Packages

For almost everything you want to do with R, there's probably a package written to do just that.

A list of packages in the official packages repository CRAN can be found here: <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 devtools library.

Working directory & 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:

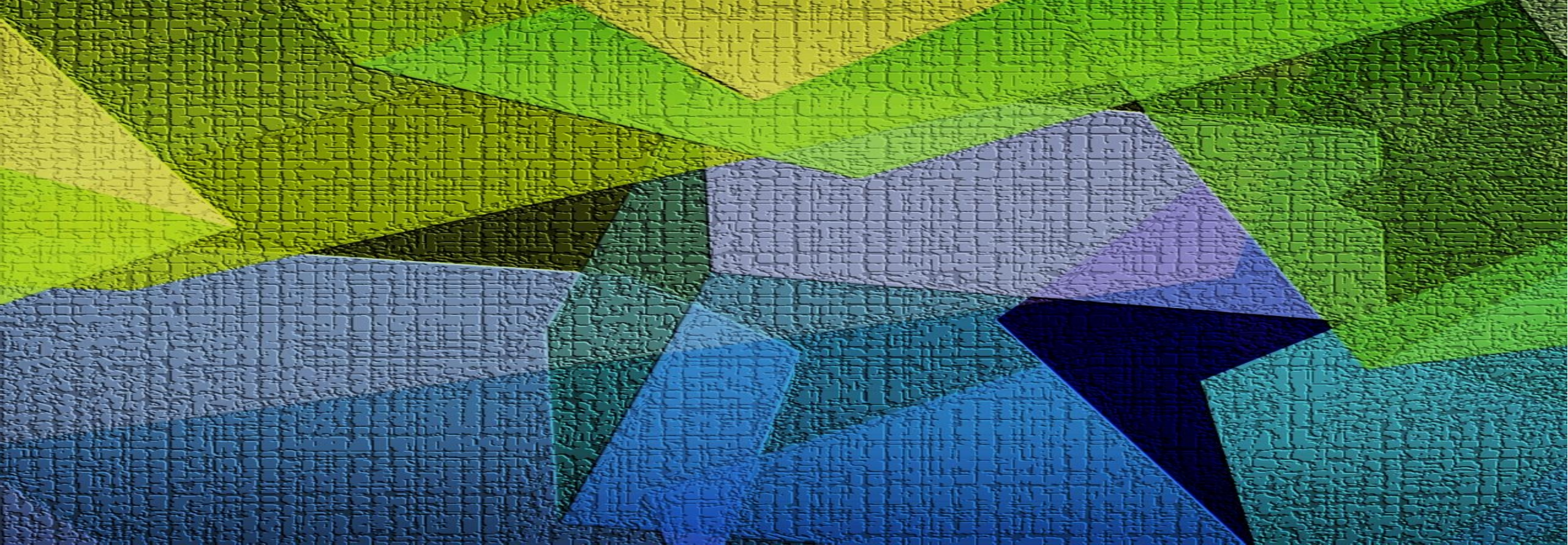
```
dir() # returns a list of  
strings of file names  
dir(pattern=".R$") # list of  
files ending in ".R"  
dir("C:\\Users") # show files in  
directory C:\\Users
```

Run a script:

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

Summary of Basic Operations

1. Subsetting
2. Control structures
3. Build-in functions
4. User written function
5. `"*apply"`
6. Packages
7. working directory and R script



R Tutorial

- Reading and Writing Data

Yuhui Zhang
yuhui@datasciencedojo.com

Raja Iqbal
raja@datasciencedojo.com
datasciencedojo
unleash the data scientist in you

Reading/writing local flat files

`read.table`, `read.csv`, and `readLines`

CSV stands for 'Comma Separated Values'

```
> 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

Reading/writing local flat files

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

write.table, write.csv, and writeLines

CSV stands for 'Comma Separated Values'

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

Reading/writing local flat files

Notice the parameters before reading/writing!

Ex.: the `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:

file, the name of a file, or a connection

header, logical indicating if the file has a header line

sep, a string indicating how the columns are separated

colClasses, a character vector indicating the class of each column in the dataset

nrows, the number of rows in the dataset

comment.char, a character string indicating the comment character

skip, the number of lines to skip from the beginning

stringsAsFactors, should character variables be coded as factors?

Reading/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)
```

Connection interfaces

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

file: opens a connection to a file

gzipfile: 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

Connection interfaces

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)
```

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>
```

Reading XML/HTML files

```
> 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'

Reading/writing JSON

JSON: Javascript object notation.

Common format for data from application programming interfaces (APIs). 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
```


Reading/writing JSON

```
> 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)
```

Connect to a data base

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      ailMell
3      allMis1
4      anoCar1
5      anoCar2
6      anoGam1
> dbDisconnect(ucscDb) # don't forget to close the connection
```

Textual format

1. With the metadata
2. Editable in the case of corruption, potentially recoverable
3. Adhere to the “Unix philosophy” (*Eric Raymond’s 17 Unix Rules*)
4. Not space-efficient

Functions to read/write textual format

dput vs. dget

dump vs. source

save vs. load

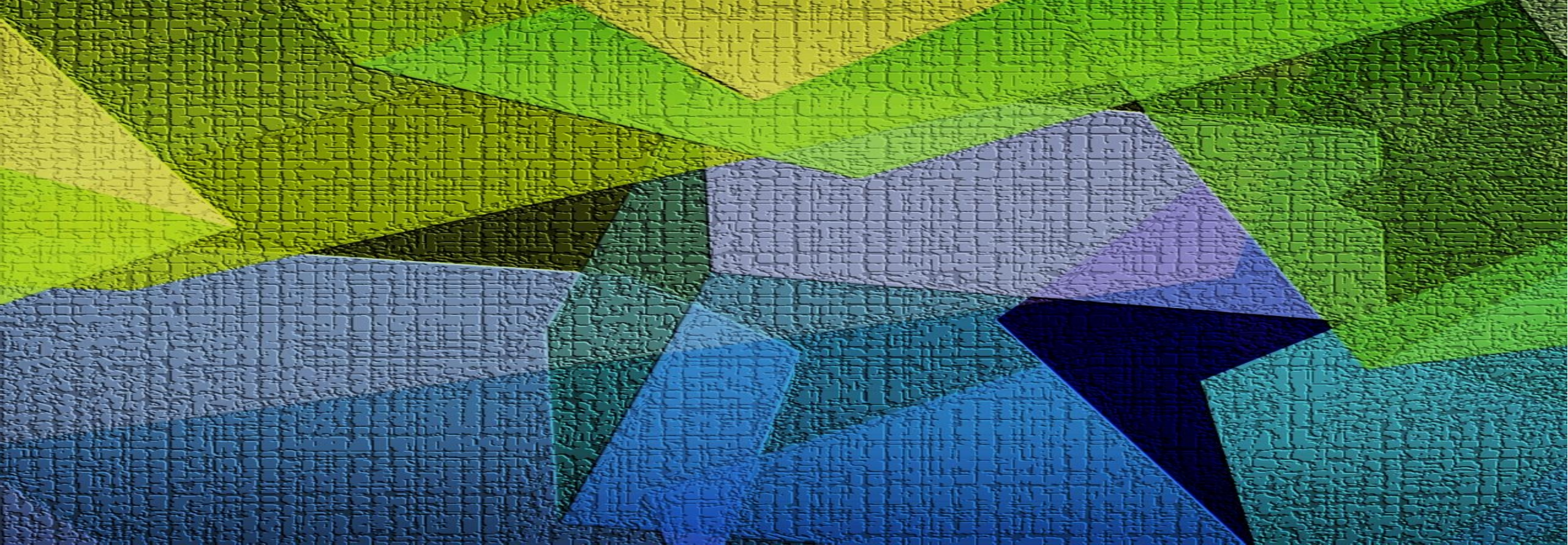
Textual format

Example:

```
save(R_object1, R_object2, file="R_objects.RData")  
load(file="R_objects.RData")
```

Summary – R/W Data

1. R/W local flat files
2. R/W local Excel files
3. Connection interfaces
4. Reading XML/HTML files
5. R/W to JSON
6. Connect to a database
7. Textual format



R Tutorial

- Statistical Simulation (optional)

Yuhui Zhang
yuhui@datasciencedojo.com

Raja Iqbal
raja@datasciencedojo.com
datasciencedojo
unleash the data scientist in you

Statistical Simulation

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

d for density, like `dnorm()`

r for random number generation, like `rnorm()`

p for cumulative distribution, like `pnorm()`

q for quantile function, like `qnorm()`

Statistical Simulation

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
```

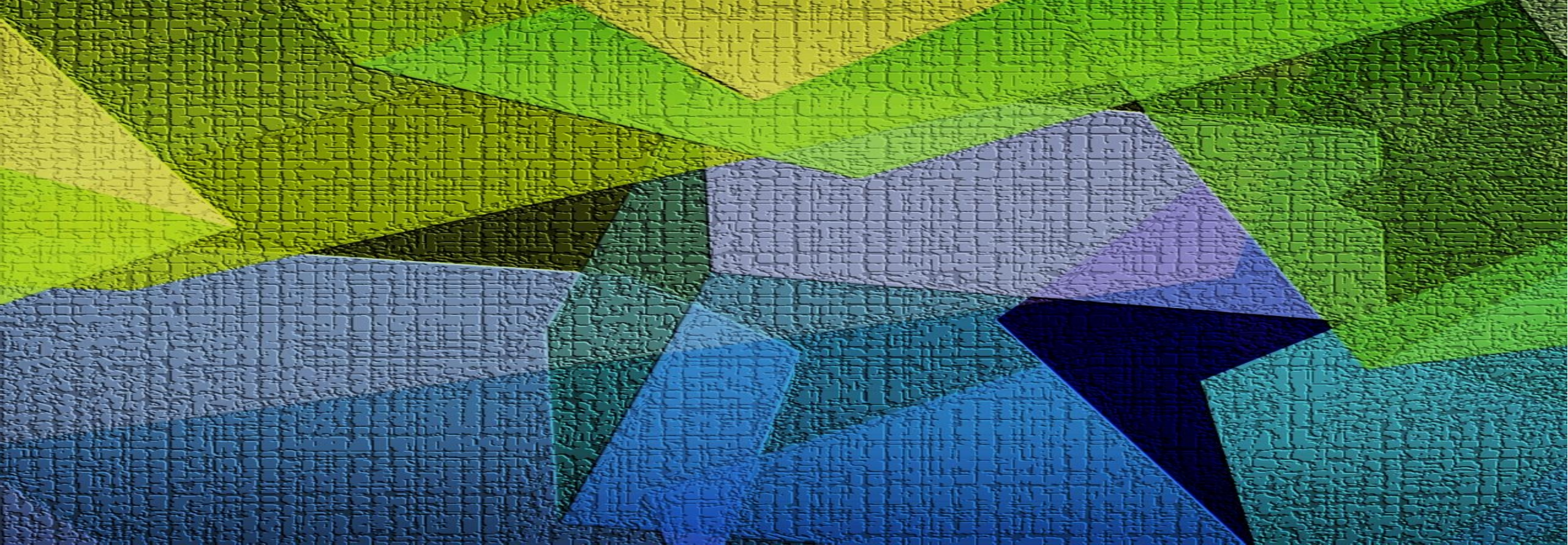

Statistical Simulation

Build-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
```

R Tutorial

- Basic Plotting Systems (optional)

Yuhui Zhang
yuhui@datasciencedojo.com

Raja Iqbal
raja@datasciencedojo.com
datasciencedojo
unleash the data scientist in you

Basic plotting systems

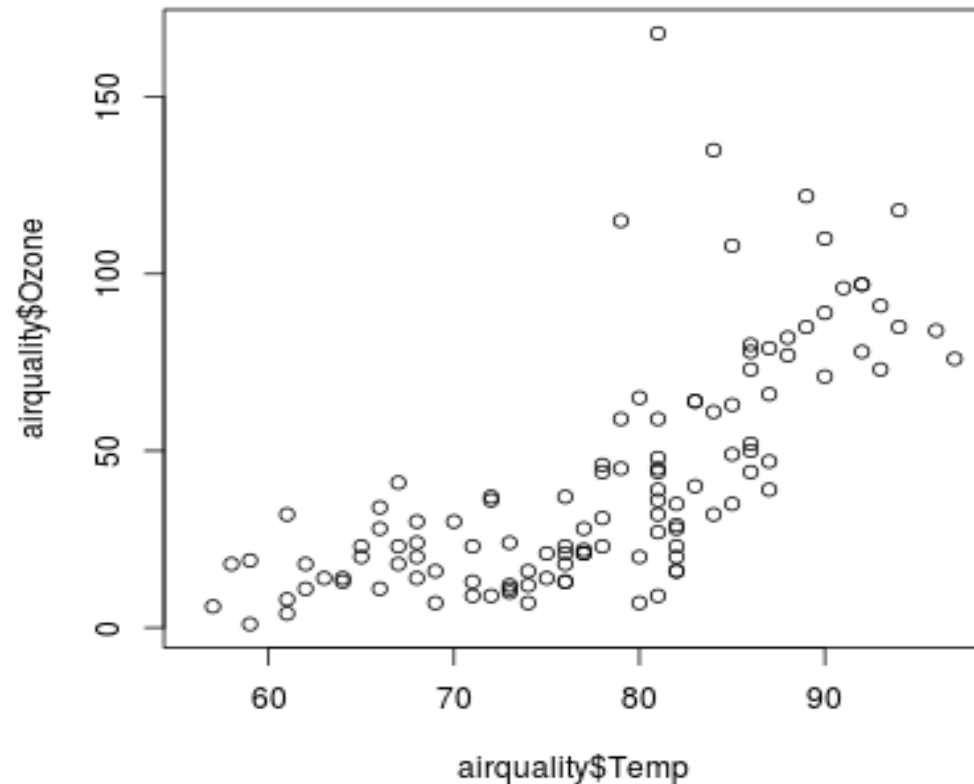
1. **Base graphics:** constructed piecemeal. Conceptually simpler and allows plotting to mirror the thought process.
2. **Lattice graphics:** entire plots created in a simple function call.
3. **ggplot2 graphics:** an implementation of the Grammar of Graphics by Leland Wilkinson. Combines concepts from both base and lattice graphics. (Need to install ggplot2 library)
4. **Fancier and more telling ones:** wait for the bootcamp!

A list of interactive visualization in R at:

<http://ouzor.github.io/blog/2014/11/21/interactive-visualizations.html>

Basic plotting systems

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

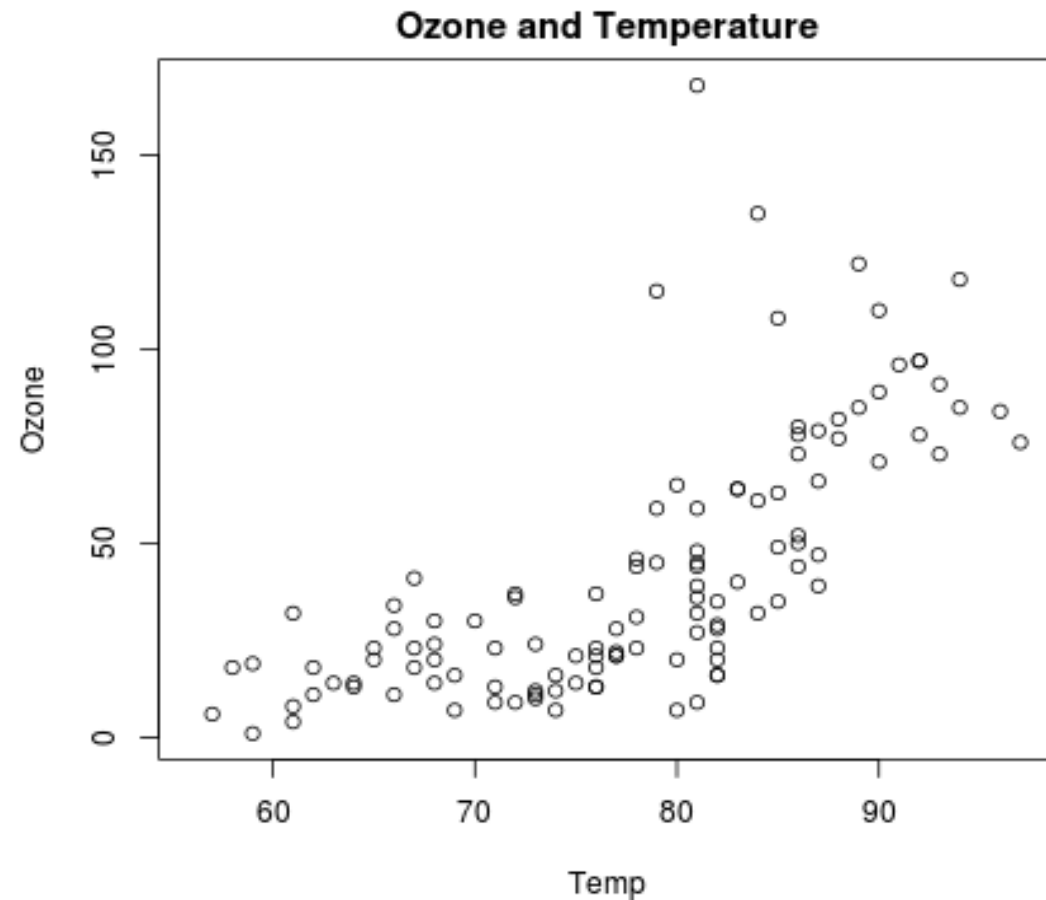
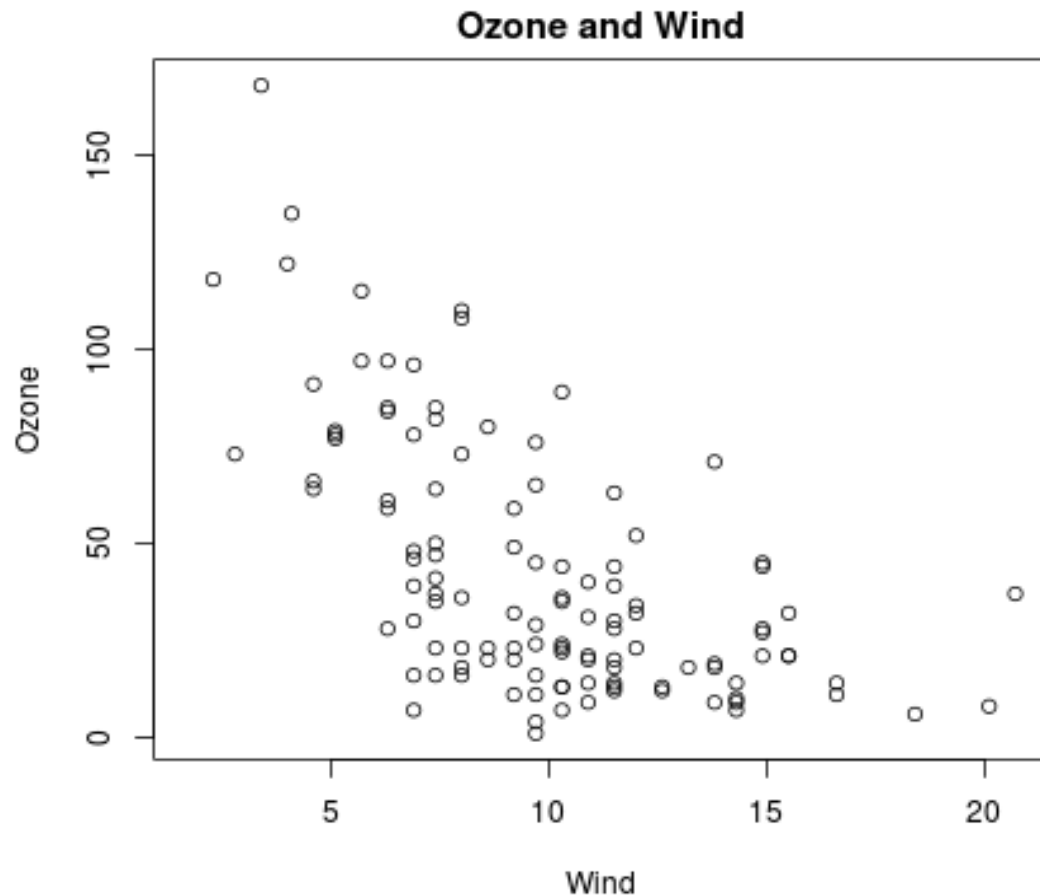


Basic plotting systems

```
# 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)
+ })
```

Basic plotting systems

Ozone and Weather in New York City



Basic plotting systems

Plotting functions

lines: adds lines to a plot, given a vector of x values and corresponding vector of y values

points: adds a point to the plot

text: add text labels to a plot using specified x,y coordinates

title: add annotations to x,y axis labels, title, subtitles, outer margin

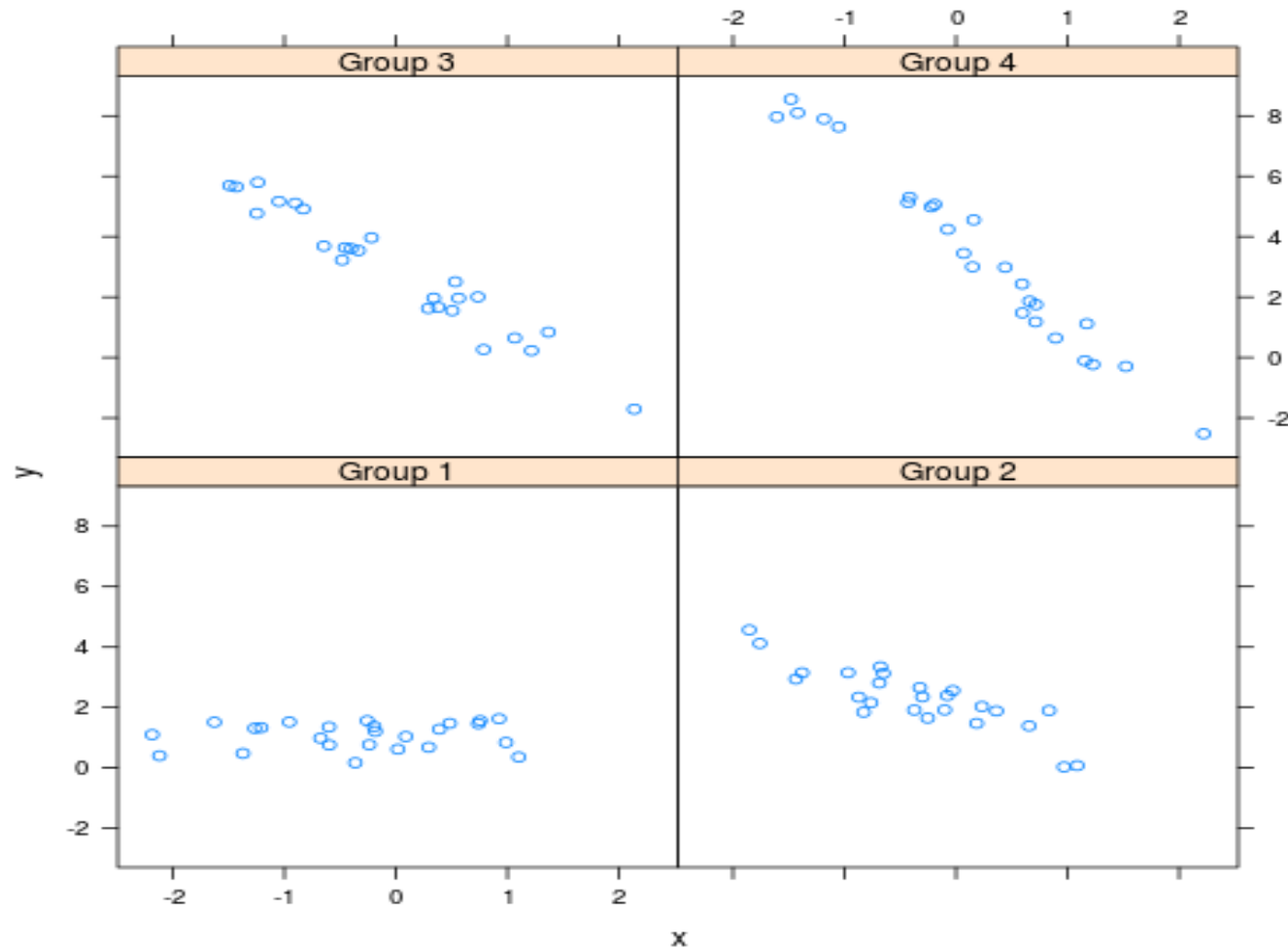
mtext: add arbitrary text to margins (inner or outer) of plot

axis: specify axis ticks

Lattice plotting systems

```
> 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)
```


Lattice plotting systems

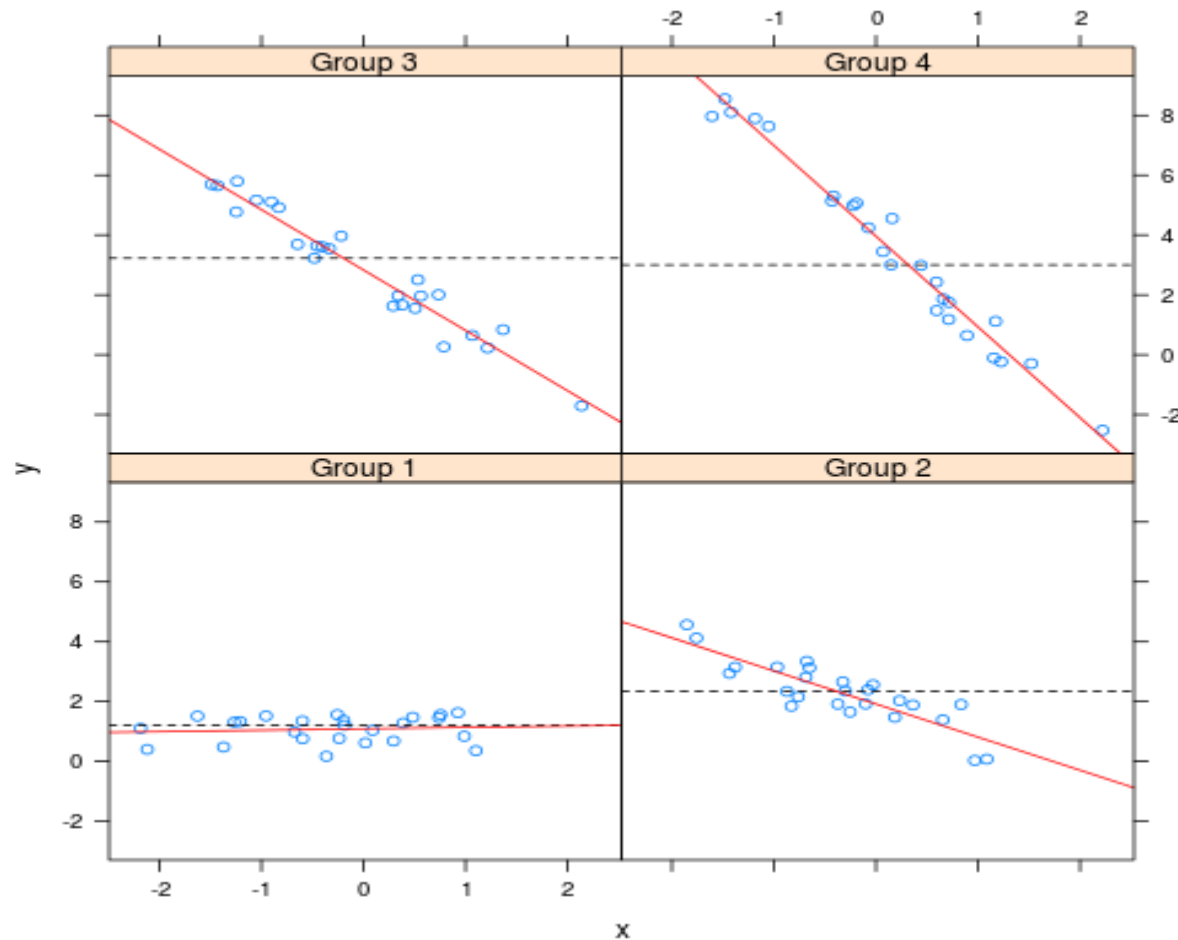


Lattice plotting systems

What more on the plot? Customize the panel function:

```
> 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)  
})
```

Lattice plotting systems



Lattice plotting systems

Plotting functions

xyplot() -> main function for creating scatterplots

bwplot() -> box and whiskers plots (box plots)

histogram() -> histograms

stripplot() -> box plot with actual points

dotplot() -> plot dots on “violin strings”

spiom() -> scatterplot matrix (like pairs() in base plotting system)

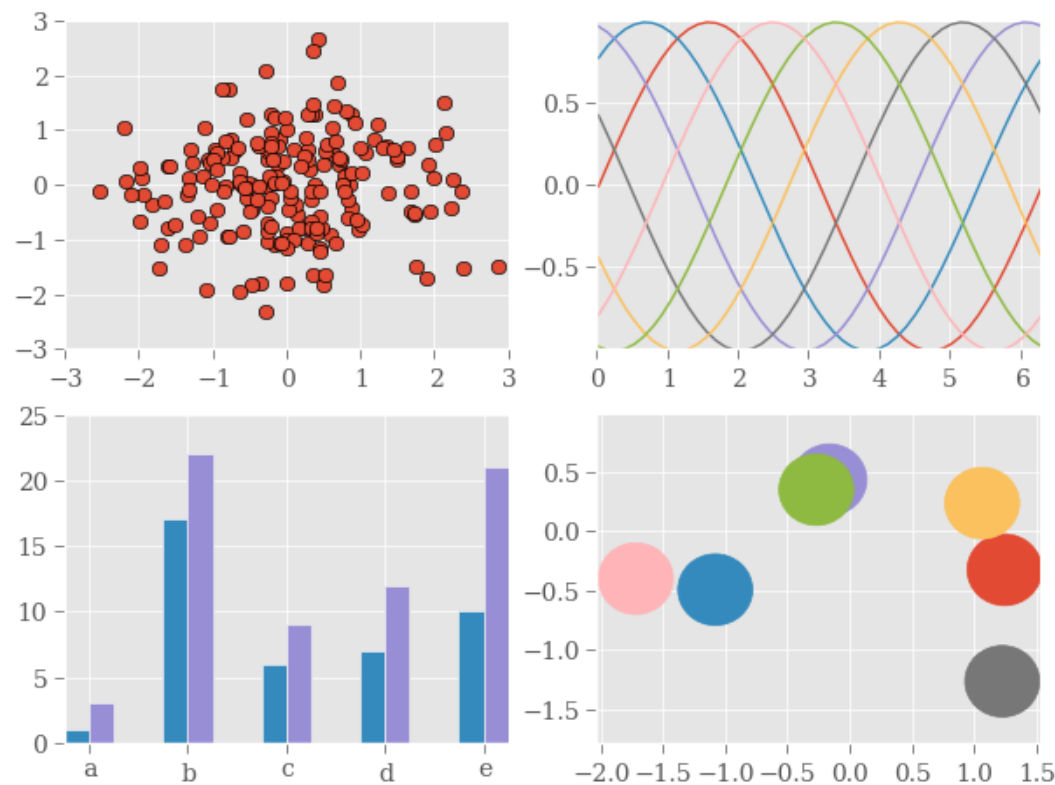
levelplot()/contourplot() -> plotting image data

ggplot2 plotting systems

Need to install ggplot2
library

Mix elements of base
and lattice

Good tutorial (3 basic
plotting systems):
[https://sux13.github.io/
DataScienceSpCourseN
otes/4_EXDATA/Explora
tory_Data_Analysis_Co
urse_Notes.html](https://sux13.github.io/DataScienceSpCourseNotes/4_EXDATA/Exploratory_Data_Analysis_Course_Notes.html)



End

But actually it's a **start!**