

# R

Bioinformatics Applications (PLPTH813)

Sanzhen Liu

2/5/2019

# Outline

- R introduction
- Data structure
- Data input and output
- Basic graphics
- String operations
- Functions
- Simple statistical test

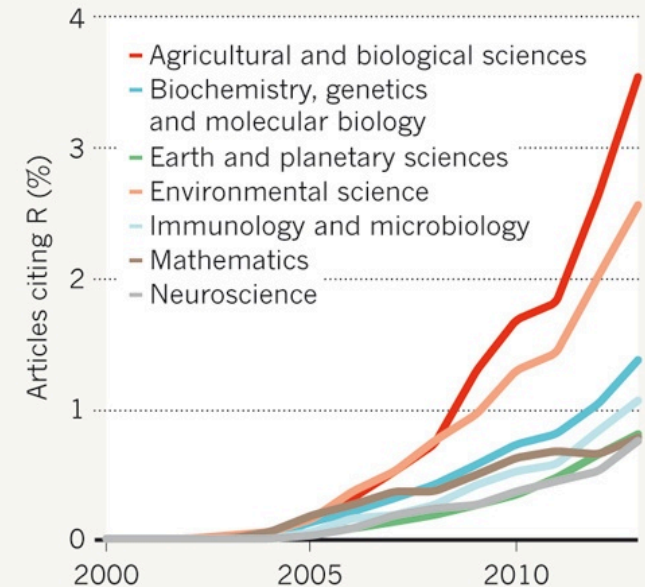
# R

- R is a programming language and a cutting-edge tool for data analysis, especially for statistical computing and graphics.
- R is free
- R is powerful. Applications are easily created by writing new **functions**. Functions are usually distributed through **packages**.
- It has great community support.

[www.r-project.org](http://www.r-project.org)

## A RISING TIDE OF R

An increasing proportion of research articles explicitly reference R or an R package.



<http://www.nature.com/news/programming-tools-adventures-with-r-1.16609#/rise>

## Example – statistical test

- $\chi^2$  test

A	B	
	12	36
	24	70

```
d <- c(12, 36, 24, 70)
dm <- matrix(d, nrow=2, byrow=T)
chisq.test(dm)
```

data: dm

X-squared = 0, df = 1, p-value = 1

```

# Christmas tree
L <- matrix(
  c(0.03, 0, 0, 0.1,
    0.85, 0.00, 0.00, 0.85,
    0.8, 0.00, 0.00, 0.8,
    0.2, -0.08, 0.15, 0.22,
    -0.2, 0.08, 0.15, 0.22,
    0.25, -0.1, 0.12, 0.25,
    -0.2, 0.1, 0.12, 0.2),
  nrow=4)
# ... and each row is a translation vector
B <- matrix(
  c(0, 0,
    0, 1.5,
    0, 1.5,
    0, 0.85,
    0, 0.85,
    0, 0.3,
    0, 0.4),
  nrow=2)

prob = c(0.02, 0.6, .08, 0.07, 0.07, 0.07, 0.07)

# Iterate the discrete stochastic map
N = 1e5 #5 # number of iterations
x = matrix(NA,nrow=2,ncol=N)
x[,1] = c(0,2) # initial point
k <- sample(1:7,N,prob,replace=TRUE) # values 1-7

for (i in 2:N)
  x[,i] = crossprod(matrix(L[,k[i]],nrow=2),x[,i-1]) + B[,k[i]] # iterate

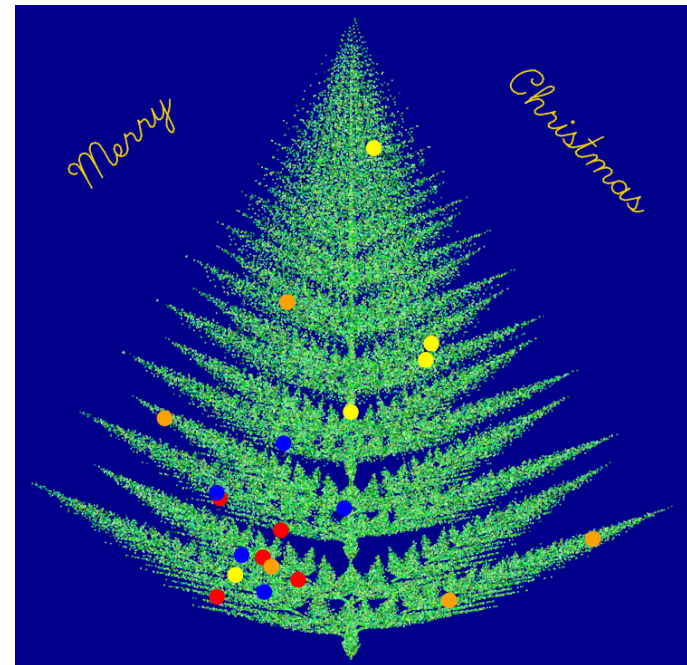
# Plot the iteration history
#png('card.png')
par(bg='darkblue',mar=rep(0,4))
plot(x=x[,1],y=x[,2],
     col=grep('green',colors(),value=TRUE),
     axes=FALSE,
     cex=.1,
     xlab='',
     ylab='')# ,pch='.')

bals <- sample(N,20)
points(x=x[,1,bals],y=x[,2,bals]-.1,
       col=c('red','blue','yellow','orange'),
       cex=2,
       pch=19)

)
text(x=-.7,y=8,
     labels='Merry',
     adj=c(.5,.5),
     srt=45,
     vfont=c('script','plain'),
     cex=3,
     col='gold')
)
text(x=0.7,y=8,
     labels='Christmas',
     adj=c(.5,.5),
     srt=-45,
     vfont=c('script','plain'),
     cex=3,
     col='gold')
)

```

## Example – Christmas tree



# R commands, case sensitivity

- **Expression:** Print the value and not save the value in the environment

`2 + 4`

`68 * 0.15`

- **Assignment:** Assign values to a **variable**

`y <- 2`

`y = 2`

`assign("y", 2)`

**Y** <- 2 + 4

- **Comments (#)**

Notes/explanation to the scripts, starting with a hashtag ('#'), everything to the end of the line is a comment.

`y <- 2 + 4 # an example of the assignment`

`y <- 2 + 4`

# Data structure – vector (I)

A vector is a single entity consisting of an ordered collection of numbers, characters, logical quantities, etc.

- Numeric vector**

```
x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

```
sum(x)
```

```
y <- 2
```

```
2*x + y
```

**c(10.4, 5.6, 3.1, 6.4, 21.7)**

↑      ↑  
1st    2nd ...

**x[2]**

- Logical vector**

```
lv <- c(TRUE, FALSE, TRUE, TRUE)
```

```
lv == FALSE
```

```
sum(lv)
```

```
# The logical operators are <, <=, >, >=, ==, and !=.
```

```
# == for exact equality and != for inequality.
```

```
x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

```
lv2 <- x > 10
```

## Data structure – vector (II)

- **Character vectors**

```
cv <- c("a", "b", "c")
```

```
cv2 <- paste(cv, 1:3, sep="")
```

- **Missing values: NA, not available**

```
mvv <- c("a", "b", "c", NA)
```

```
is.na(mvv)
```



# Select a subset and modify a vector

- **Select a subset of a vector**

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

```
x[c(2, 3)]
```

```
x[x>10]
```

```
x[-c(1,5)]
```

- **Modify a vector**

```
x[3] <- 23.1
```

```
x <- c(x, 10.9)
```

```
names(x) <- c("a", "b", "c", "d", "e", "f")
```

# mode and length of a vector

- **Mode**

Vectors must have their values with the same mode, either numeric, character, logical, or other types.

```
z <- 0:9
```

```
is.numeric(z)
```

```
digits <- as.character(z) # convert to character
```

```
d <- as.integer(digits) # convert to integer
```

- **Length**

```
length(z)
```

```
length(z) <- 5 # retain just the first 5 values
```

# factor

Definition: A factor is a vector object used to specify a discrete classification (grouping) of the components of other vectors with the same length.

- **factor = regular vector + Levels**

```
state <- c("tas", "sa", "qld", "nsw",  
          "nsw", "nt", "wa", "wa",  
          "qld", "vic", "nsw", "vic")  
statef <- factor(state)  
> statef  
[1] tas sa qld nsw nsw nt wa wa qld vic nsw vic  
Levels: nsw nt qld sa tas vic wa  
> levels(statef)  
[1] "nsw" "nt" "qld" "sa" "tas" "vic" "wa"
```

```
state2 <- as.character(statef)
```

# array and matrix

- array: 2 or more dimensions of data
- matrix: a special array with two dimensions

```
num <- 1:25  
numm <- matrix(num, nrow=5, byrow=T)  
nrow(numm)  
dim(numm)
```

matrices can be built up by using the functions `cbind()` and `rbind()`:

**`cbind()`** forms matrices by binding together matrices horizontally, or column-wise  
**`rbind()`** vertically, or row-wise.

Note: the result of `rbind()` or `cbind()` always has matrix status

# data.frame

name	age	>30?	gender
Josh	23	FALSE	male
Rose	35	TRUE	female
Jone	18	FALSE	male
Molly	21	FALSE	female
Lisa	36	TRUE	female

- **Data frame**

A data frame may be regarded as a matrix with columns possibly of differing modes and attributes. The data of a matrix are of the same type or mode.

- **Making data frames**

```
df <- data.frame(name=c("Josh", "rose"), age=c(23, 35))
```

- **Working with data frames**

```
> df$name
[1] Josh rose
Levels: Josh rose
> df[, 1]
[1] Josh rose
Levels: Josh rose
```

```
> df[[1]]
[1] Josh rose
Levels: Josh rose
> df[1]
  name
1 Josh
2 rose
```

```
head(df); tail(df); summary(df); str(df)
```

# list

A list is an object consisting of an ordered collection of objects.

- **Objects can be any types or modes**

```
lst <- list(name="Fred", wife="Mary", nkids=3, kid.ages=c(4,7,9))
```

```
> lst[1] # sublist
```

```
$name
```

```
[1] "Fred"
```

```
> lst[[1]] # first element in the list
```

```
[1] "Fred"
```

```
> lst$name # the element named "name"
```

```
[1] "Fred"
```

# Problem

```
df <- data.frame(name=c("Josh", "rose", "John"),  
age=c(23, 35, 18))
```

What are the values of

`df[2, 1]`

`df[3, 2]`

`df[2]`

`df[, 2]`

name	age
Josh	23
Rose	35
Jone	18

What is the difference between the last two?

# Data import

- **scan()**: to read data from a file to a vector or list

```
cat("lisa Jone", "28 21", file = "hrdb.txt", sep = "\n")  
hr <- scan("hrdb.txt", what=character())  
hr  
"lisa" "Jone" "28" "21"
```

- **read.table()**: to read a data frame (table) directly

read.delim, read.csv

```
d <- read.table(data)
```

Input file form with names and row labels:

	Price	Floor	Area	Rooms	Age	Cent.heat
01	52.00	111.0	830	5	6.2	no
02	54.75	128.0	710	5	7.5	no
03	57.50	101.0	1000	5	4.2	no
04	57.50	131.0	690	6	8.8	no
05	59.75	93.0	900	5	1.9	yes
...						



# Data export

- **write.table()** or **write.csv()**

```
## To write a tab-delimited file:
```

```
x <- data.frame(a = "pi", b = pi)
```

```
write.table(x, file="foo.txt", sep="\t", row.names=FALSE)
```

```
## and to read this file back into R one needs
```

```
read.table("foo.txt")
```

```
## Alternatively
```

```
write.csv(x, file = "foo.csv", row.names=FALSE)
```

```
read.csv("foo.csv")
```

# Outline

- R introduction
- Data structure
- Data input and output
- **Basic graphics**
- **String operations**

# Basic graphics

- **plot(); points(); lines(); abline(); text(); legend()**

**High-level plot: create a new plot**

```
plot(x, y, xlab, ylab, main, ...)
```

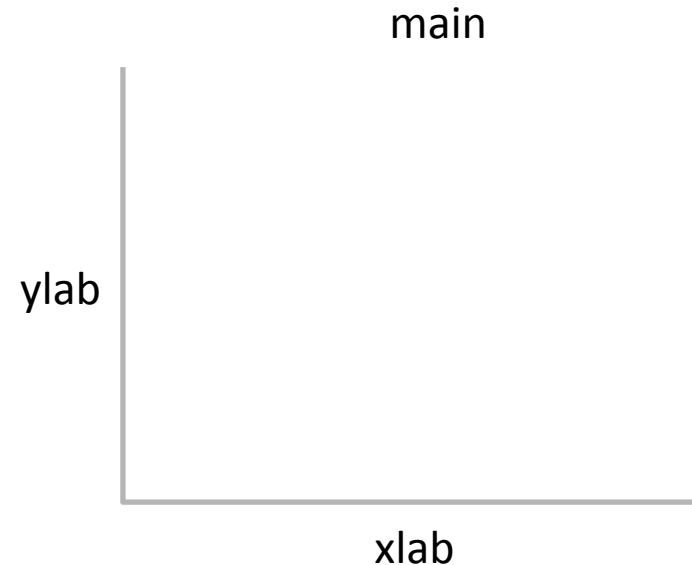
**Low-level plot: add to an existing plot**

```
# add points  
points(x, y)
```

```
# add lines  
lines(x, y)
```

```
# add horizontal or vertical lines  
abline(h, v)
```

```
# add text or legend  
text()  
legend()
```



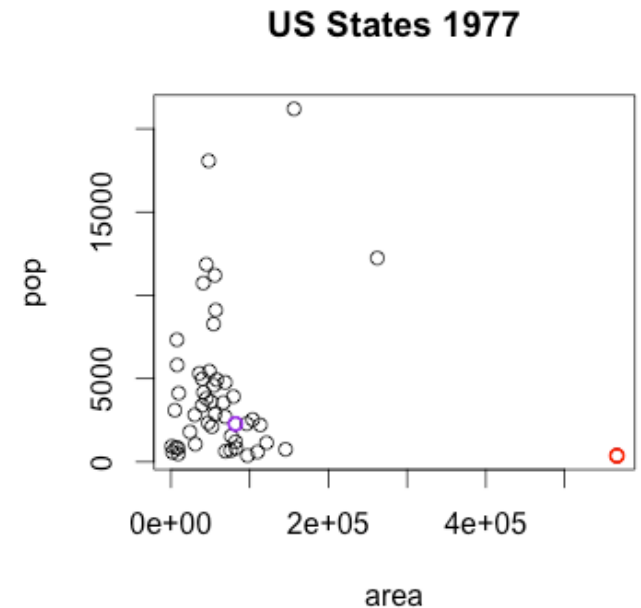
# Scatter plot

```
# data
area <- state.x77[, "Area"]
pop <- state.x77[, "Population"]

# scatter plot
plot(area, pop, main="US States 1977")

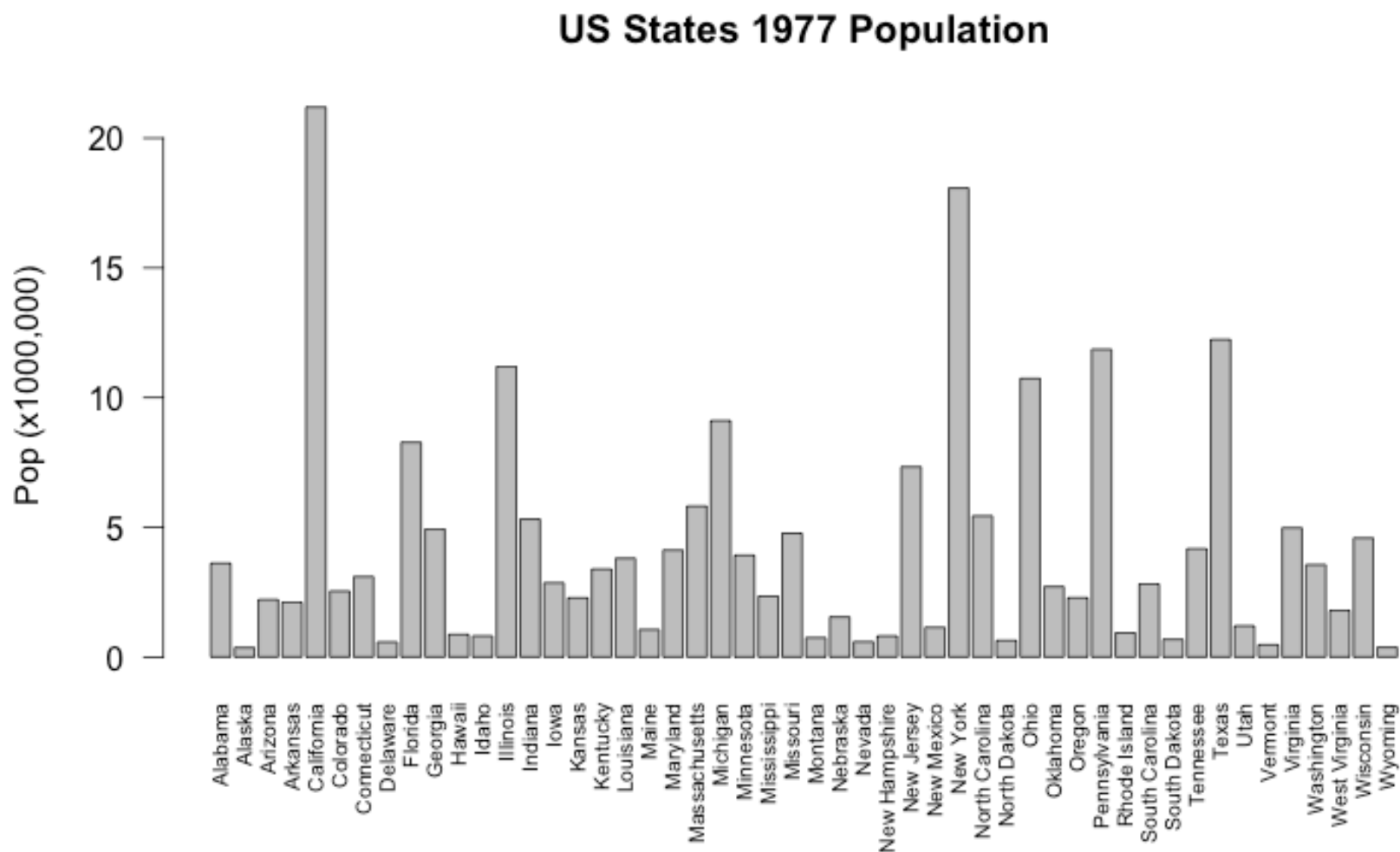
# label points
state.max.area <- which.max(area)
points(area[state.max.area],
       pop[state.max.area],
       col="red", lwd=2)

points(area["Kansas"], pop["Kansas"],
       col="purple", lwd=2)
```

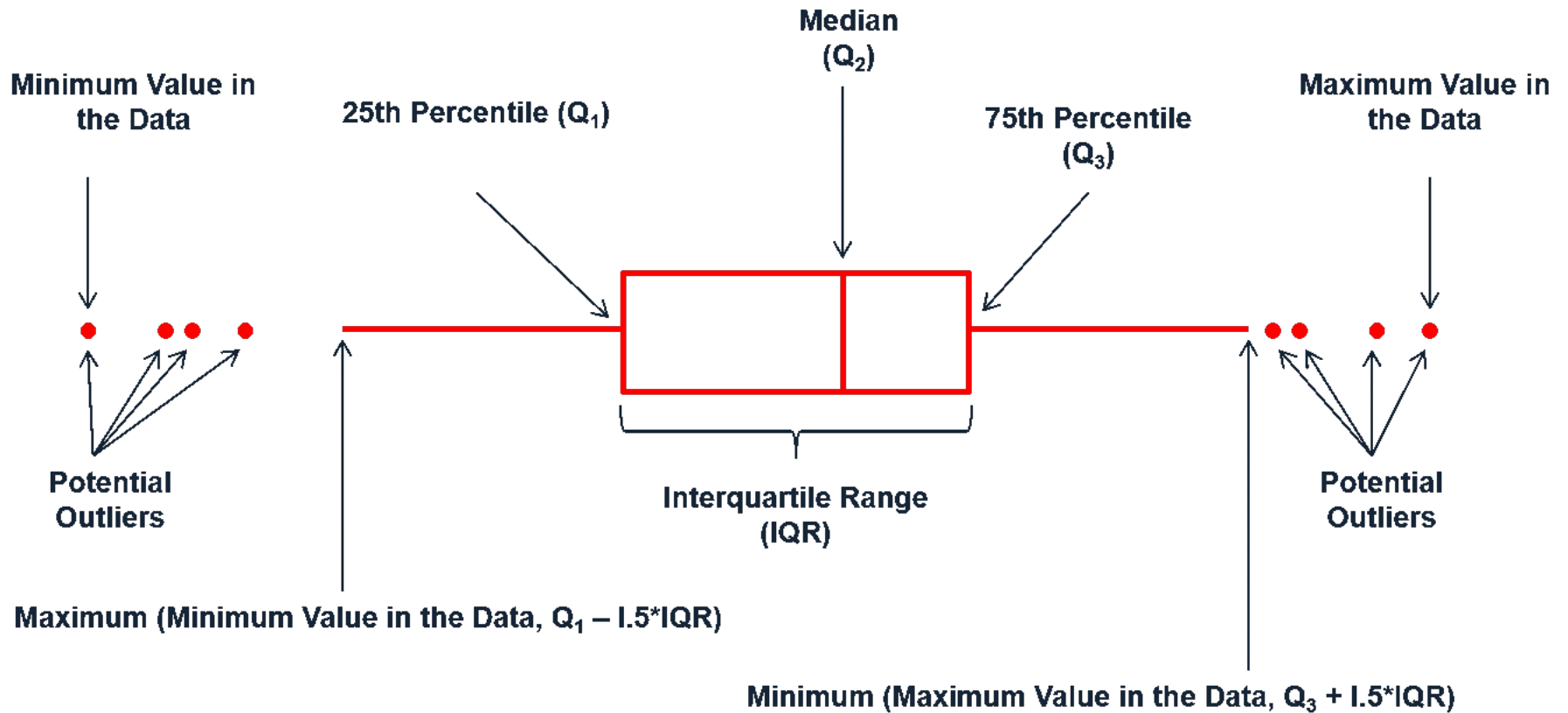


# Barplot

```
barplot(pop/1000, las=2, cex.names=0.65, ylab="Pop (x1000,000)",  
        main="US States 1977 Population")
```

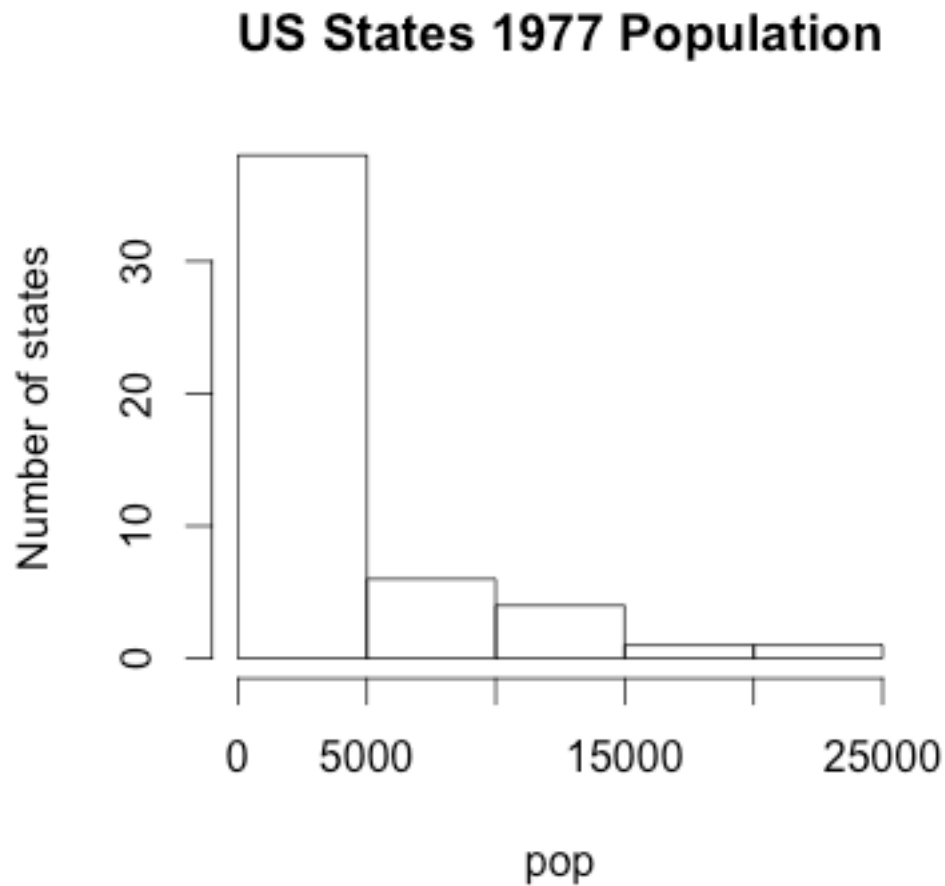


# Boxplot



# Histogram

```
hist(pop, ylab="Number of states", main="US States 1977 Population")
```



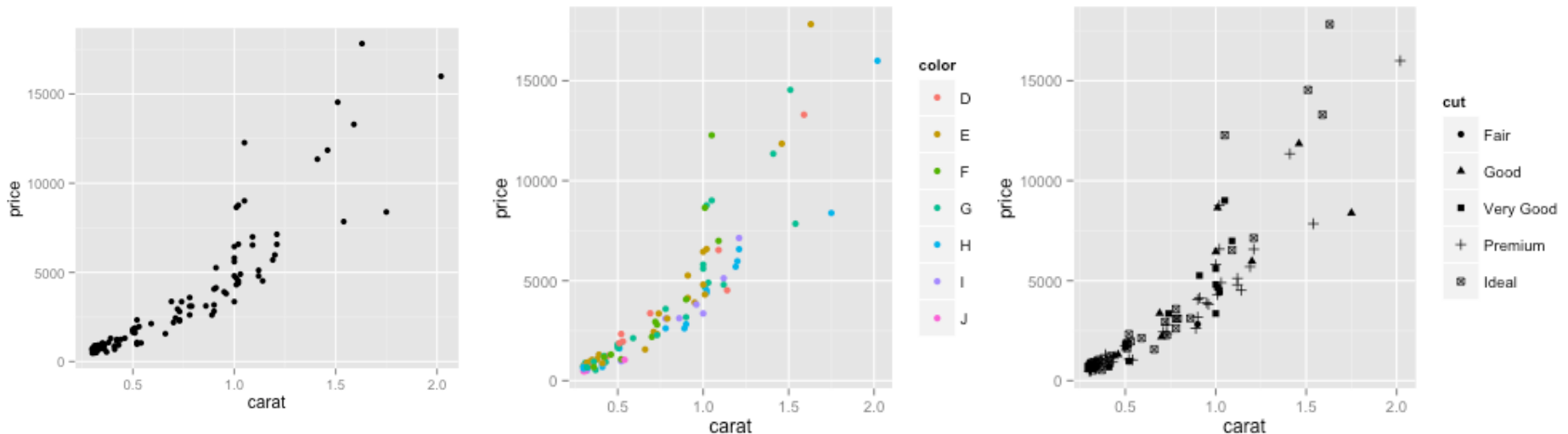
# ggplot2 - an easy plotting package

diamonds

carat	cut	color	clarity	depth	table	price
0.23	Ideal	E	SI2	61.5	55	326
0.21	Premium	E	SI1	59.8	61	326

**scatterplots** showing the relationship between the price and carats (weight) of a diamond\*.

```
qplot(carat, price, data = diamonds)
qplot(carat, price, data = diamonds, colour = color)
qplot(carat, price, data = diamonds, shape = cut)
```



\* from <http://ggplot2.org/book/qplot.pdf>



## ggplot2 - geom to control plot type

qplot is not limited to scatterplots, but can produce almost any kind of plot by varying the **geom**. geom has many options:

- "point" draws a scatterplot. This is the default.
- "smooth" fits a smoother to the data
- "boxplot" produces a box-and-whisker plot
- "line" draw lines between the data points.
- "histogram" draws a histogram
- "bar" makes a bar chart

# ggplot2 – a flexible tool to plot various plots

diamonds

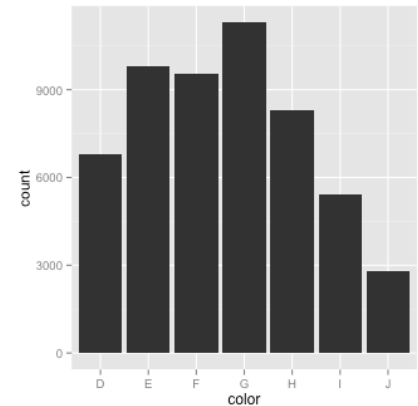
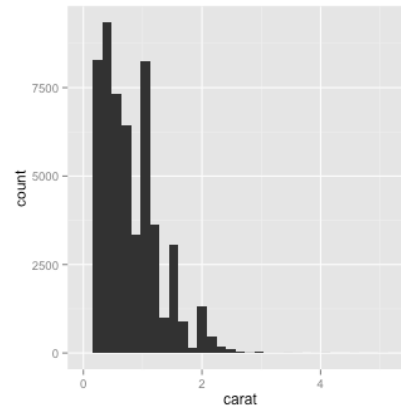
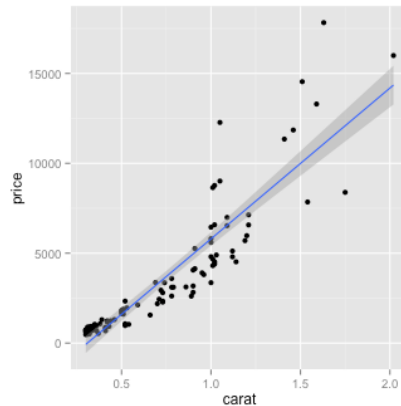
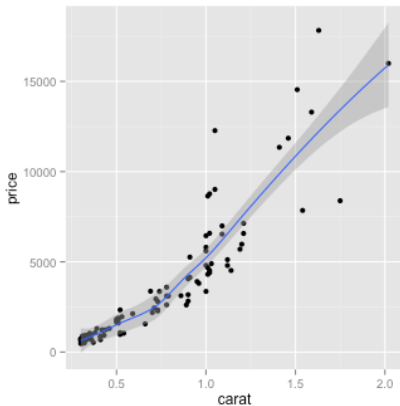
carat	cut	color	clarity	depth	table	price
0.23	Ideal	E	SI2	61.5	55	326
0.21	Premium	E	SI1	59.8	61	326

## Adding a smooth line or a fitted line

```
qplot(carat, price, data = diamonds, geom = c("point", "smooth"))  
qplot(carat, price, data = diamonds, geom = c("point", "smooth"),  
      method = "lm")
```

## Histogram and barplot

```
qplot(carat, data = diamonds, geom = "histogram")  
qplot(color, data = diamonds, geom = "bar")
```



## String operations - nchar

- **nchar()**

nchar the sizes of the corresponding elements of a vector.

nchar(cvec)

```
> cvec
```

```
[1] "google" "hello"  "the"    "world"
```

```
> nchar(cvec)
```

```
[1] 6 5 3 5
```

## String operations - grep

- **grep()**

grep searches for matches to argument pattern within each element of a character vector

`grep("o", cvec)`

```
> cvec  
[1] "google" "hello"  "the"    "world"
```

```
> grep("o", cvec)  
[1] 1 2 4
```

## String operations – sub and gsub

- **sub()** and **gsub()**

sub and gsub perform replacement of the *first* and *all* matches respectively.

```
sub("o", "O", cvec)
```

```
gsub("o", "O", cvec)
```

```
> cvec
```

```
[1] "google" "hello"  "the"    "world"
```

```
> sub("o", "O", cvec)
```

```
[1] "gOogle" "hellO"  "the"    "wOrld"
```

```
> gsub("o", "O", cvec)
```

```
[1] "gOOgle" "hellO"  "the"    "wOrld"
```

# Outline

- R introduction
- Data structure
- Data input and output
- Basic graphics
- String operations
- **Functions**
- Simple statistical test

# function/module in R

- If a procedure is repeated multiple times, it would be valuable to convert the procedure to a function/module.

- **Define a function**

```
fun_name <- function(arg_1, arg_2, ...) expression
```

or

```
fun_name <- function(arg_1, arg_2, ...) {  
  expressions  
}
```

- **Use a function**

```
fun_name(arg_1, arg2, ...)
```

## Function example 2

- **Define a function**

name <- function(arg\_1, arg\_2, ...) expression

```
# example 1
```

```
threetimes <- function(x) {  
  y <- 3*x  
  y  
}
```

```
> threetimes(6)  
[1] 18
```

```
> val <- threetimes(29)  
> val  
[1] 87
```



## Function example 2

```
# return the value of the nth element of the input vector
what_at_n <- function(in_vector, n) {
  # initiate the output value
  nth_val <- NA

  if (n <= length(in_vector)) {
    nth_val <- in_vector[n]
  }

  print_info <- paste("The value of element", n, "is", nth_val, sep=" ")
  print(print_info)
  nth_val
}
```

```
> what_at_n(c(36, 19, 13), 2)
[1] "The value of element 2 is 19"
[1] 19
```

```
> val2 <- what_at_n(c(36, 19, 13), 2)
[1] "The value of element 2 is 19"
> val2
[1] 19
```

## base (build-in) functions in R

- R has many build-in functions
- What we learn is to know how to use them
- If you have choices to use a build-in function, do not use your own function (efficiency and code sharing)

# "apply" functions

- **apply**
- **lapply**
- **sapply**
- **mapply**
- **tapply**
  
- vapply
- rapply
- ...

goal: to simplify coding and improve computation efficiency

# apply()

- **apply(X, MARGIN, FUN, ...)**

apply a function to margins of an array or matrix.

d

3.95	3.98	2.43	10.36
3.89	3.84	2.31	10.04
4.05	4.07	2.31	10.43
4.2	4.23	2.63	11.06
4.34	4.35	2.75	11.44
3.94	3.96	2.48	10.38

`apply(d, 1, sum)`

`apply(d, 2, sum)`    24.37    24.43    14.91

**rowSums**

**colSums**

# apply - example

```
> head(diamonds)
  carat      cut color clarity depth table price     x     y     z
1  0.23    Ideal     E    SI2   61.5     55   326  3.95  3.98  2.43
2  0.21  Premium     E    SI1   59.8     61   326  3.89  3.84  2.31
3  0.23     Good     E    VS1   56.9     65   327  4.05  4.07  2.31
4  0.29  Premium     I    VS2   62.4     58   334  4.20  4.23  2.63
5  0.31     Good     J    SI2   63.3     58   335  4.34  4.35  2.75
6  0.24 Very Good     J   VVS2   62.8     57   336  3.94  3.96  2.48
```

```
> apply(diamonds[, c("carat", "price")], 2, mean)
      carat      price
0.7979397 3932.7997219
```

## combine your own function with apply

```
sumsqrt <- function(x) {  
  sum(sqrt(x))  
}  
apply(d, 1, sumsqrt)
```

or

```
apply(d, 1, function(x) sum(sqrt(x)))
```

3.95	3.98	2.43	5.54
3.89	3.84	2.31	5.45
4.05	4.07	2.31	5.55
4.2	4.23	2.63	5.73
4.34	4.35	2.75	5.83
3.94	3.96	2.48	5.55



# sapply and lapply

## `sapply()` and `lapply()`

work in a similar way, calling the specified function for each item of a list or vector.

```
> sapply(1:3, function(x) x^2)
[1] 1 4 9
```

## `lapply` returns a list rather than a vector:

```
> lapply(1:3, function(x) x^2)
[[1]]
[1] 1
```

```
[[2]]
[1] 4
```

```
[[3]]
[1] 9
```

# mapply

## **mapply()**

vectorize arguments to a function that is not usually accepting vectors as arguments.

```
> rep(1:3, 3)
[1] 1 2 3 1 2 3 1 2 3
```

```
> mapply(rep, 1:3, 3)
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
```

```
> mapply(rep, 1:3, 3:1)
[[1]]
[1] 1 1 1
[[2]]
[1] 2 2
[[3]]
[1] 3
```

1. apply each element from the 3<sup>rd</sup> argument to each element in the 2<sup>nd</sup> argument using the function specified in the 1<sup>st</sup> argument
2. combine them by column or organize them in a data frame or list format



# tapply

- **tapply()**

Applying a function to each element of a vector given by the category of each element, provided by the other vector.

```
> head(diamonds)
  carat      cut color clarity depth table price     x     y     z
1  0.23    Ideal     E    SI2   61.5     55   326  3.95  3.98  2.43
2  0.21  Premium     E    SI1   59.8     61   326  3.89  3.84  2.31
3  0.23     Good     E    VS1   56.9     65   327  4.05  4.07  2.31
4  0.29  Premium     I    VS2   62.4     58   334  4.20  4.23  2.63
5  0.31     Good     J    SI2   63.3     58   335  4.34  4.35  2.75
6  0.24 Very Good     J   VVS2   62.8     57   336  3.94  3.96  2.48
```

```
> tapply(diamonds$price, diamonds$cut, mean)
      Fair      Good Very Good   Premium      Ideal
4358.758  3928.864  3981.760  4584.258  3457.542
```

# aggregate

## **aggregate(X, by, FUN, ...)**

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
6	0.24	Very Good	J	VVS2	62.8	57	336	3.94	3.96	2.48

```
> aggregate(diamonds$price, by=list(diamonds$cut), FUN=mean)
```

	Group.1	x
1	Fair	4358.758
2	Good	3928.864
3	Very Good	3981.760
4	Premium	4584.258
5	Ideal	3457.542

```
> tapply(diamonds$price, diamonds$cut, FUN=mean)
```

Fair	Good	Very Good	Premium	Ideal
4358.758	3928.864	3981.760	4584.258	3457.542

# table

- **table()**

Determining counts for each category

```
> head(diamonds)
  carat      cut color clarity depth table price     x     y     z
1  0.23    Ideal     E    SI2   61.5     55   326  3.95  3.98  2.43
2  0.21  Premium     E    SI1   59.8     61   326  3.89  3.84  2.31
3  0.23     Good     E    VS1   56.9     65   327  4.05  4.07  2.31
4  0.29  Premium     I    VS2   62.4     58   334  4.20  4.23  2.63
5  0.31     Good     J    SI2   63.3     58   335  4.34  4.35  2.75
6  0.24 Very Good     J   VVS2   62.8     57   336  3.94  3.96  2.48
```

```
> table(diamonds$cut)
```

Fair	Good	Very Good	Premium	Ideal
1610	4906	12082	13791	21551

# Outline

- R introduction
- Data structure
- Data input and output
- Basic graphics
- String operations
- Functions
- **Simple statistical test**

# t-test

## t.test

Performs one and two sample t-tests on vectors of data.

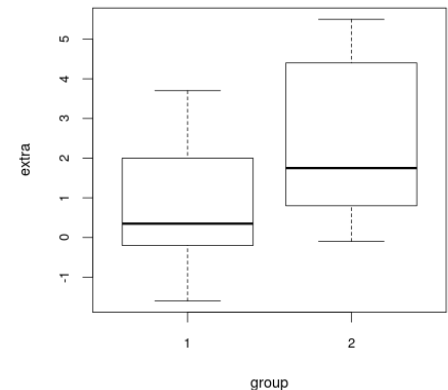
```
# Student's sleep data
plot(extra ~ group, data = sleep)

# t-test
with(sleep, t.test(extra[group == 1],
extra[group == 2]))
```

```
# Formula
t.test(extra ~ group, data = sleep)
```

data: sleep

extra	group	ID
0.7	1	1
-1.6	1	2
-0.2	1	3
-1.2	1	4
-0.1	1	5
3.4	1	6
3.7	1	7
0.8	1	8
0.0	1	9
2.0	1	10
1.9	2	1
0.8	2	2
1.1	2	3
0.1	2	4
-0.1	2	5
4.4	2	6
5.5	2	7
1.6	2	8
4.6	2	9
3.4	2	10



# Linear models (I)

## Fitting a linear model

`lm(formula, data = data.frame)`

```
pc <- lm(price ~ carat, data=diamonds)
summary(pc)
```

Residuals:

Min	1Q	Median	3Q	Max
-18585.3	-804.8	-18.9	537.4	12731.7

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-2256.36	13.06	-172.8	<2e-16 ***
carat	7756.43	14.07	551.4	<2e-16 ***

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1549 on 53938 degrees of freedom

Multiple R-squared: 0.8493, Adjusted R-squared: 0.8493

F-statistic: 3.041e+05 on 1 and 53938 DF, p-value: < 2.2e-16

# ANOVA (I)

## ANOVA

`anova(model)`

```
pcc <- lm(price ~ carat + cut, data=diamonds)
anova(pcc)
```

Analysis of Variance Table

Response: price

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
carat	1	7.2913e+11	7.2913e+11	319162.11	< 2.2e-16 ***
cut	4	6.1332e+09	1.5333e+09	671.17	< 2.2e-16 ***
Residuals	53934	1.2321e+11	2.2845e+06		

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

# ANOVA (II)

## Comparing two models

```
anova(model1, model2)
```

```
pc <- lm(price ~ carat, data=diamonds)
pcc <- lm(price ~ carat + cut,
data=diamonds)
anova(pc, pcc)
```

Analysis of Variance Table

Model 1: price ~ carat

Model 2: price ~ carat + cut

	Res.Df	RSS	Df	Sum of Sq	F	Pr(>F)
1	53938	1.2935e+11				
2	53934	1.2321e+11	4	6133201436	671.17	< 2.2e-16 ***

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1



# chi-square test

## chisq.test

```
d <- c(12, 36, 24, 70)
dm <- matrix(d, nrow=2, byrow=T)
chisq.test(dm)
```

data: dm

X-squared = 0, df = 1, p-value = 1

A	B	
	12	36
	24	70

# Online resources

## "apply" function family

- <https://www.datacamp.com/community/tutorials/r-tutorial-apply-family#gs.YUI=Luc>

## Statistical modeling with R

- <https://www.datacamp.com/courses/statistical-modeling-in-r-part-1>
- <http://www.analyticsforfun.com/2014/06/performing-anova-test-in-r-results-and.html>

## Get help

- `help(ls)`
- `?ls`
- `??colsum`: ambiguous search
- [R reference card](#)
- `stackoverflow`
- Google is the best helper!

R learning: <http://swirlstats.com/>

# Rstudio

Rstudio is an open source integrated development environment (IDE) for R

- On your own machine (Rstudio Desktop)

Download and install [R](#)

Download and install [Rstudio](#)

- Use Rstudio at Beocat (Rstudio server)

**rstudio.beocat.cis.ksu.edu**

Your KSU ID and password to login

# Adventures with R

