UCLA Department of Statistics
Statistical Consulting Center

R Bootcamp - 2010
Intermediate R

Colin Rundel
crundel@stat.ucla.edu

September 20, 2010

Part I

Subsetting Review

We have seen how to use brackets [] to subset. R has 5 ways to select specific elements from (most) objects.

1. Indexing by position
2. Indexing by exclusion
3. Indexing by name
4. Indexing by logical mask
5. Empty subsetting

Subsetting by position works by selecting elements by their numerical index. Note that R starts its indexes from 1 and not 0 like some other languages.

```
> x = c(1, 1, 2, 3, 5, 8)
> x[1]
[1] 1

> x[6]
[1] 8

> x[c(2, 3)]
[1] 1 2

> x[1:6]
[1] 1 1 2 3 5 8
```

Subsetting by exclusion works by excluding elements by their numerical index.

```
> x = c(1, 1, 2, 3, 5, 8)
> x[-1]
[1] 1 2 3 5 8

> x[-6]
[1] 1 1 2 3 5

> x[-c(2, 3)]
[1] 1 3 5 8

> x[-(1:6)]
numeric(0)
```

Subsetting by name selects elements by matching their names (if they exist, more on this later).

```
> x = c(a = 1, b = 2, c = 3)
> x["a"]
a
1

> x["b"]
b
2

> x[c("b", "c")]
b c
2 3
```

Subsetting by logical mask works by selecting elements using a logical vector of the same length where TRUE indexes are included and FALSE indexes excluded.

```
> x = c(1, 1, 2, 3, 5, 8)
> x[c(TRUE, TRUE, FALSE, FALSE, TRUE, TRUE)]
[1] 1 1 5 8

> x[c(TRUE, FALSE)]
[1] 1 2 5

> x == 1
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE

> x[x == 1]
[1] 1 1

> x[x%%2 == 0]
[1] 2 8
```

Empty subsetting selects all elements. This is useful when using subsetting for assignment.

```
> x = c(1, 2, 3)
> y = c(1, 2, 3)
> x[]
[1] 1 2 3

> x = 3
> y[] = 3
> x
[1] 3

> y
[1] 3 3 3
```

It is possible to use subsetting with assignment to change values of only specific elements.

```
> x = c(1, 1, 2, 3, 5, 8)
> x[1] = 8
> x
[1] 8 1 2 3 5 8

> x[1:4] = c(-1, -2)
> x
[1] -1 -2 -1 -2  5  8

> x[x%%2 != 0] = 0
> x
[1]  0 -2  0 -2  0  8
```

Part II

Data Classes

Every R object has a number of attributes.
Some of the most important are:

- mode: Mutually exclusive classification of objects according to their basic structure.
  - logical, integer, double, complex, raw, character, list, expression, function, NULL, ...
- class: Property assigned to an object that determines how generic functions operate with it. If no specific class is assigned to object, by default it is the same as the mode.
  - vector, matrix, array, data.frame, list, factor, ...

Mode / Type	Definition
logical	boolean value that is either TRUE or FALSE
numeric	numerical value can either be an integer or floating point (double)
complex	complex numerical value
character	character string
function	collection of arbitrary R expressions

Class	Definition
vector	1-d array of elements of the same class
matrix	2-d array of elements of the same class
array	n-d array of elements of the same class
data.frame	2-d array of elements where elements in the same column have the same class
list	Generic vector where elements can be of any class
factor	Categorical variable with defined levels

R allows for automatic type coercion. This is usually helpful but can also cause problems.

```
> x = c(1, 2, 3, 8)
> y = rep(TRUE, 3)
> c(x, y)
[1] 1 2 3 8 1 1 1

> c(x, "a")
[1] "1" "2" "3" "8" "a"

> c(y, "a")
[1] "TRUE" "TRUE" "TRUE" "a"
```

Problematic coercion:

```
> z = factor(c("A", "A", "B", "A"))
> z
[1] A A B A
Levels: A B

> c(z, 1)
[1] 1 1 2 1 1

> c(z, FALSE)
[1] 1 1 2 1 0

> c(z, "A")
[1] "1" "1" "2" "1" "A"
```

Explicit coercion is also possible within R usually using "as" functions:

```
> as.numeric("151")
[1] 151

> as.complex("1")
[1] 1+0i

> as.factor(c("h", 1, "3"))
[1] h 1 3
Levels: 1 3 h

> as.character(c(1, 2, 3))
[1] "1" "2" "3"
```

Lengths of objects are also subject to implicit coercion (this is some times known as recycling).

For many operations on two objects if one object is shorter than the other, then the elements of the shorter object are repeated to produce an object of the same length as the longer one.

```
> 1:4 + 1
[1] 2 3 4 5

> 1:4 + c(2, 3)
[1] 3 5 5 7

> 1:4 + 1:3
[1] 2 4 6 5
```

```
> 1:4 + c(1, 1, 1, 1)
[1] 2 3 4 5

> 1:4 + c(2, 3, 2, 3)
[1] 3 5 5 7

> 1:4 + c(1, 2, 3, 1)
[1] 2 4 6 5
```

Special Objects / Classes:

Function	Test Function	Definition
NA	is.na	Represents missing data
NULL	is.null	Represents the null / empty object
NaN	is.nan	Represents a numerical value that is not a number
Inf	is.infinite	Represents an infinite value

```
> x = NA
> x == 1
[1] NA

> x == NA
[1] NA

> is.na(x)
[1] TRUE
```

```
> x = c()
> x == 1
logical(0)

> x == NULL
logical(0)

> is.null(x)
[1] TRUE
```

Character objects are immutable strings.

Unlike other languages there are no direct ways of accessing the characters that make up the string.

```
> length("text")
[1] 1

> nchar("text")
[1] 4
```

To create or modify character vectors the most useful function is *paste* which concatenates string arguments

```
> paste("X", "Y")
[1] "X Y"

> paste("X", "Y", sep = " + ")
[1] "X + Y"

> paste("Fig", 1:4)
[1] "Fig 1" "Fig 2" "Fig 3" "Fig 4"

> paste(c("X", "Y"), 1:4, sep = "", collapse = " + ")
[1] "X1 + Y2 + X3 + Y4"
```

Other Common Character Functions:

Function	Definition
substr	Extracts or replaces a substring
strsplit	Splits string at specific patterns
toupper	All characters in string to upper case
tolower	All characters in string to lower case
grep	Regex string matching
gsub	Regex string replacement

Matrices are an extension of a vector into 2 dimensions. There are several approaches available to construct a matrix.

```
> matrix(1:9, ncol = 3, nrow = 3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> matrix(1:9, ncol = 3, nrow = 3, byrow = TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Matrices may be of any mode, but all elements must have the same mode.

```
> matrix(c(TRUE, FALSE, TRUE), ncol = 3, nrow = 3)
      [,1]  [,2]  [,3]
[1,]  TRUE  TRUE  TRUE
[2,] FALSE FALSE FALSE
[3,]  TRUE  TRUE  TRUE

> matrix(rep(c("AB", "MN", "YZ"), 3), ncol = 3, nrow = 3)
     [,1] [,2] [,3]
[1,] "AB" "AB" "AB"
[2,] "MN" "MN" "MN"
[3,] "YZ" "YZ" "YZ"
```

Matrices can also be built by adding on a vector/matrix on to an existing vector/matrix by row or column.

```
> rbind(1:3, 4:6, 7:9)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9

> cbind(1:3, 4:6, 7:9)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

What would *m* look like if the following code was run?

```
> m = matrix(c(1, 2), 3, 2, byrow = TRUE)
```

What would *m* look like if the following code was run?

```
> m = matrix(c(1, 2), 3, 2, byrow = TRUE)
```

```
> m
     [,1] [,2]
[1,]    1    2
[2,]    1    2
[3,]    1    2
```

What about *m2* if the following code was run?

```
> m2 = cbind(rbind(m, 1:2), 3)
```

What about *m2* if the following code was run?

```
> m2 = cbind(rbind(m, 1:2), 3)
```

```
> m2
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
[4,]    1    2    3
```

Common Matrix Functions:

Function	Definition
dim	dimensions of the matrix
ncol	number of columns
nrow	number of rows
colSums	calculates the sum of columns
rowSums	calculates the sum of rows
t	produces the transpose of the matrix
%*%	matrix multiplication operator
solve	calculates inverse of the matrix

Be careful when using a function that does not explicitly take a matrix as an argument as the results can be unpredictable.

```
> m2
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
[4,]    1    2    3
```

```
> sum(m2)
[1] 24

> mean(m2)
[1] 2

> sd(m2)
[1] 0 0 0
```

Some functions have assignment methods which can be used to manipulate the passed arguments. For example the *dim* function can be used to alter the dimensions of a matrix.

```
> m2
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
[4,]    1    2    3

> dim(m2) = c(2, 6)
> m2
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    2    2    3    3
[2,]    1    1    2    2    3    3
```

Arrays are the same as matrices except that they can have an arbitrary number of dimensions.

```
> (a = array(1:4, c(1, 4, 3)))
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4

, , 2

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4

, , 3

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

```
> a[, , 3]
[1] 1 2 3 4

> a[, 1:2, ]
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2

> a[1, , ]
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3
[4,]    4    4    4
```

In the second and third examples from the last slide, R is dropping the third dimension from the array this behavior can be suppressed by the *drop* argument.

```
> a[, 1:2, ]
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
```

```
> a[, 1:2, , drop = FALSE]
, , 1

     [,1] [,2]
[1,]    1    2

, , 2

     [,1] [,2]
[1,]    1    2

, , 3

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

A data frame is a 2-d array of data where only the columns are constrained to be of the same type. This is the default data class used when reading in data from a file.

```
> scores = read.csv("scores.csv")
> scores
   Name     id Quiz1 Exam1 Exam2 Quiz2 Exam3
1 Susan 123412    50    47    33    67    79
2  John 548963    38    61    75    59    65
3   Bob 234563    89    97    85    88    92
4  Bill 429591    72    73    74    75    76
5  Mary 245887    92    95    79    89    90
6  Paul  97522    99     3    55    60    72
```

With this data set there was a header row, R uses the elements in this row to name the columns in the data frame. We can use these names to access the columns using *$* or *[[ ]]*.

```
> scores$Name
[1] Susan John  Bob   Bill  Mary  Paul
Levels: Bill Bob John Mary Paul Susan

> scores$Quiz2
[1] 67 59 88 75 89 60

> scores[["Exam1"]]
[1] 47 61 97 73 95  3

> scores[["id"]]
[1] 123412 548963 234563 429591 245887  97522
```

We can also use traditional indexing to access elements, rows, and columns in a data frame.

```
> scores[, 1]
[1] Susan John   Bob    Bill   Mary   Paul
Levels: Bill Bob John Mary Paul Susan

> scores[2, ]
  Name      id Quiz1 Exam1 Exam2 Quiz2 Exam3
2 John 548963    38    61    75    59    65

> scores[3, 3:7]
  Quiz1 Exam1 Exam2 Quiz2 Exam3
3    89    97    85    88    92
```

If you wanted a list of names and ids of all students who scored under 50 on one of the three exams?

If you wanted a list of names and ids of all students who scored under 50 on one of the three exams?

```
> scores[, c(4, 5, 7)] < 50
     Exam1 Exam2 Exam3
[1,]  TRUE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,]  TRUE FALSE FALSE
```

If you wanted a list of names and ids of all students who scored under 50 on one of the three exams?

```
> scores[, c(4, 5, 7)] < 50
     Exam1 Exam2 Exam3
[1,]  TRUE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,]  TRUE FALSE FALSE
```

```
> as.logical(rowSums(scores[, c(4, 5, 7)] < 50))
[1]  TRUE FALSE FALSE FALSE FALSE  TRUE
```

If you wanted a list of names and ids of all students who scored under 50 on one of the three exams?

```
> scores[, c(4, 5, 7)] < 50
     Exam1 Exam2 Exam3
[1,]  TRUE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,]  TRUE FALSE FALSE
```

```
> as.logical(rowSums(scores[, c(4, 5, 7)] < 50))
[1]  TRUE FALSE FALSE FALSE FALSE  TRUE
```

```
> scores[as.logical(rowSums(scores[, c(4, 5, 7)] < 50)), c(1, 2)]
   Name     id
1 Susan 123412
6  Paul  97522
```

Data frames can be created/modified in much the same way as matrices.

```
> (d = data.frame(a = c(1, 2, 3), b = c("m", "n", "o"), c = TRUE))
  a b    c
1 1 m TRUE
2 2 n TRUE
3 3 o TRUE

> d$d = factor(c("a", "b", "c"))
> d[, 5] = as.complex(1)
> cbind(d, f = 1)
  a b    c d   V5 f
1 1 m TRUE a 1+0i 1
2 2 n TRUE b 1+0i 1
3 3 o TRUE c 1+0i 1
```

Lists are a generic vector that allows for the collection of arbitrary objects / classes.

```
> (l = list(a = c(TRUE, FALSE), b = matrix(1:4, 2, 2), "hello"))
$a
[1]  TRUE FALSE

$b
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
[1] "hello"
```

Elements of lists can be accessed using *$* and or *[[ ]]*.

```
> l$a
[1]  TRUE FALSE

> l[["b"]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> l[[3]]
[1] "hello"
```

[ ] can also be used to select one or more elements, but the returned object will be a list.

```
> l[3]
[[1]]
[1] "hello"

> class(l[3])
[1] "list"

> l["a"]
$a
[1]  TRUE FALSE

> class(l["a"])
[1] "list"
```

```
> l[c(1, 2)]
$a
[1]  TRUE FALSE

$b
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> class(l[c(1, 2)])
[1] "list"
```

Names of elements can also be altered after the fact using the *names* replacement function.

```
> names(l)
[1] "a" "b" ""

> names(l)[3] = "c"
> l[["c"]]
[1] "hello"

> names(l) = c("x", "y", "z")
> names(l)
[1] "x" "y" "z"

> l[["x"]]
[1]  TRUE FALSE
```

The *names* function can also be used with any of the other data classes we have discussed so far.

```
> a = c(1, 2, 3, 4)
> names(a) = c("w", "x", "y", "z")
> a
w x y z
1 2 3 4

> a["x"]
x
2

> a[c("w", "y")]
w y
1 3
```

In the case of matrices you can label columns and rows via the *colnames* and *rownames* functions.

```
> b = matrix(1:4, 2, 2)
> colnames(b) = c("x", "y")
> rownames(b) = c("m", "n")
> b
  x y
m 1 3
n 2 4

> b[, "x"]
m n
1 2

> b["n", ]
x y
2 4
```

Remember the data frame we used earlier?

```
> scores
   Name     id Quiz1 Exam1 Exam2 Quiz2 Exam3
1 Susan 123412    50    47    33    67    79
2  John 548963    38    61    75    59    65
3   Bob 234563    89    97    85    88    92
4  Bill 429591    72    73    74    75    76
5  Mary 245887    92    95    79    89    90
6  Paul  97522    99     3    55    60    72
```

With objects that have a name attribute you can use the *attach* or the *with* command.

```
> attach(scores)
> mean(Exam1 + Exam2 + Exam3)
[1] 208.5

> Name
[1] Susan John  Bob   Bill  Mary  Paul
Levels: Bill Bob John Mary Paul Susan

> detach(scores)
> with(scores, mean(Exam1 + Exam2 + Exam3))
[1] 208.5
```

Note - *attach* should never be used as it can result namespace collisions.

Part III

Defining Functions

Functions in R are defined using the *function* keyword. Curly braces are used to define the body of the function.

The value / object returned by the function is indicated by the *return* keyword.

```
> square = function(x) {
+     return(x^2)
+ }
> square(5)
[1] 25

> square(1:3)
[1] 1 4 9
```

If the function's code is only one line then the braces are not needed (this is true for anywhere curly braces are used)

return calls are also implicit if not present, the output of the last expression in the function is returned.

```
> cube = function(x) x^3
> cube(4)
[1] 64

> cube(1:3)
[1]  1  8 27
```

Arguments for functions can be given default values using =

When calling a function arguments can be explicitly referenced by name otherwise ordering is used.

```
> pow = function(x, y = 2) x^y
> pow(2)
[1] 4

> pow(2, 4)
[1] 16

> pow(y = 4, 2)
[1] 16

> pow(y = 3, x = 3)
[1] 27
```

In some cases it is desirable to not explicitly define the arguments a function takes, for example the *sum* function. This can be accomplished by using ... in the function definition.

It is also possible to return multiple values by combining them in a list object.

```
> mini.summary = function(...) {
+     n = c(...)
+     return(list(mean = mean(n), median = median(n)))
+ }
> mini.summary(1, 2, 2, 3)
$mean
[1] 2

$median
[1] 2
```

... can also be used with explicitly named arguments.

```
> test = function(x, y, ...) {
+     dots = paste(c(...), collapse = " ")
+     print(paste("x=", x, " y=", y, " ...=", dots, sep = ""))
+ }
> test(1, 2, 3, 4, 5, 6, 7)
[1] "x=1 y=2 ...=3 4 5 6 7"

> test(1, 2, 3, 4)
[1] "x=1 y=2 ...=3 4"

> test(1, 2, 3)
[1] "x=1 y=2 ...=3"
```

Arguments contained in ... can also be named.

```
> test2 = function(x, ...) {
+     l = list(...)
+     n = names(l)
+     for (i in n[n != ""]) {
+         cat(paste(i, "=", l[i]), "\n")
+     }
+ }
> test2(1, 2, z = 3, y = 4)
z = 3
y = 4

> test2(1, 2, mean = 3)
mean = 3
```

Changes made to variables within a given scope (function) are lost when the scope finishes.

It is possible to modify variables outside of the local scope by using the global assignment operator <<-.

```
> x = 3
> (function() x = 2)()
> x
[1] 3

> (function() x <<- 2)()
> x
[1] 2

> (function() x)()
[1] 2
```

General Advice:

- Write your own functions, (re)use them everywhere
- Writing functions will make you a better programmer
- If you find yourself copy and pasting blocks of code, write a function instead
- Read other people's functions/code (R makes this tremendously easy)
- Do not reinvent the wheel, particularly if that wheel is in base

Part IV

Flow Control and Loops

Symbol	Meaning
!	logical NOT
&	logical AND
|	logical OR
$<$	less than
$<=$	less than or equal to
$>$	greater than
$>=$	greater than or equal to
$==$	logical equals
$! =$	not equal
xor(x,y)	exclusive OR
isTRUE(x)	equivalent to x==TRUE

Basic flow control in R is accomplished with *if else* commands.

```
> even.odd = function(x) {
+     if (!is.numeric(x)) {
+         print("neither")
+     }
+     else if (x%%2 == 0) {
+         print("even")
+     }
+     else {
+         print("odd")
+     }
+ }
```

```
> even.odd(3)
[1] "odd"

> even.odd(4)
[1] "even"

> even.odd("A")
[1] "neither"

> even.odd(NA)
[1] "neither"

> even.odd(c(3, 4))
[1] "odd"
```

if does not work with vectors, only the first element is used. To test logical vectors the *any* and *all* functions can be used.

```
> any(3 == 1:5)
[1] TRUE

> any(0 == 1:5)
[1] FALSE

> all(1:5 == 1:5)
[1] TRUE

> all(1 == 1:2)
[1] FALSE
```

ifelse is a function with similar use, it returns a specified value if the test is TRUE or a second specified value if it is FALSE. (This function handles vector arguments properly)

```
> ifelse(3%%2 == 0, "even", "odd")
[1] "odd"

> ifelse(4%%2 == 0, "even", "odd")
[1] "even"

> ifelse(c(3, 4)%%2 == 0, "even", "odd")
[1] "odd"  "even"
```

There are three main types of loops in R, *for*, *while* and *repeat*.

for loops are defined by a variable that iterates over a set of elements.

```
> for (x in 1:3) {
+     print(x)
+ }
[1] 1
[1] 2
[1] 3

> for (x in c("hello", "goodbye")) {
+     print(x)
+ }
[1] "hello"
[1] "goodbye"
```

for loops will iterate over just about any basic object. (Not that I suggest you do this)

```
> m = matrix(1:4, nrow = 2, ncol = 2)
> for (x in m) print(x)
[1] 1
[1] 2
[1] 3
[1] 4

> d = data.frame(a = c(1, 2), b = "A")
> for (x in d) print(x)
[1] 1 2
[1] A A
Levels: A

> l = list(a = c(1, 2), b = c("A"))
> for (x in d) print(x)
[1] 1 2
[1] A A
Levels: A
```

Behavior within loops can be modified using the *next* and *break* commands.

Within a *for* loop:

- *next* moves the iterator to the next element and continues at the start of the loop
- *break* immediately exits the loop

```
> for (x in 1:9) {
+     if (x%%2 == 0)
+         next
+     if (x == 7)
+         break
+     print(x)
+ }
[1] 1
[1] 3
[1] 5
```

A *while* loop repeats until the given condition becomes false.

- *next* forces the loop back to the start
- *break* immediately exits the loop

```
> x = 1
> while (x < 3) {
+     print(x)
+     x = x + 1
+ }
[1] 1
[1] 2
```

A *repeat* loop is equivalent to a *while* loop where the condition is always true, the only way to exit is by using break.

- *next* forces the loop back to the start.

```
> x = 1
> repeat {
+     print(x)
+     x = x + 1
+     if (x > 3)
+         break
+ }
[1] 1
[1] 2
[1] 3
```

Why is this output different from the *while* loop on the last slide?

In R there is a family of *apply* functions that applies a given function to each element of a vector, matrix, list, etc. These functions are usually much faster than equivalent implementations using loops.

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

```
> (x = matrix(1:9, 3, byrow = T))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9

> apply(x, 1, mean)
[1] 2 5 8

> apply(x, 2, function(x) sum(x)/length(x))
[1] 4 5 6
```

lapply and *sapply* are similar functions that work with vectors, lists, and data frames.

Both are functions are nearly identical but *sapply* simplifies the results when possible.

```
> sapply(scores[, 3:7], mean)
   Quiz1    Exam1    Exam2    Quiz2    Exam3
73.33333 62.66667 66.83333 73.00000 79.00000

> sapply(scores[, 3:7], sd)
   Quiz1    Exam1    Exam2    Quiz2    Exam3
24.68738 35.04093 19.39502 13.31165 10.43072
```

Conventional wisdom and Advice

- Looping in R is slow
- Speed mostly depends on what is occurring inside the loop(s) and the size of your data
- R has power functional programming features, use them when you can
- Most important factor - working/running code
- Premature optimization is the root of all evil
- In general: vectorization is faster than apply is faster than a loop

Part V

Additional Resources

CRAN

`http://cran.stat.ucla.edu/`

The Comprehensive R Archive Network

CRAN
[Mirrors]
[What's new?]
[Task Views]
[Search]

About R
[R Homepage]

Software
[R Sources]
[R Binaries]
[Packages]
[Other]

Documentation
[Manuals]
[FAQs]
[Contributed]
[Newsletter]

Frequently used pages

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Linux
- MacOS X
- Windows

Source Code for all Platforms

Windows and Mac users most likely want the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- **The latest release** (2008-12-22): R-2.8.1.tar.gz (read what's new in the latest version).
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

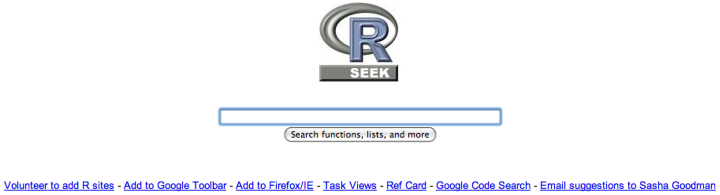
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is `GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for

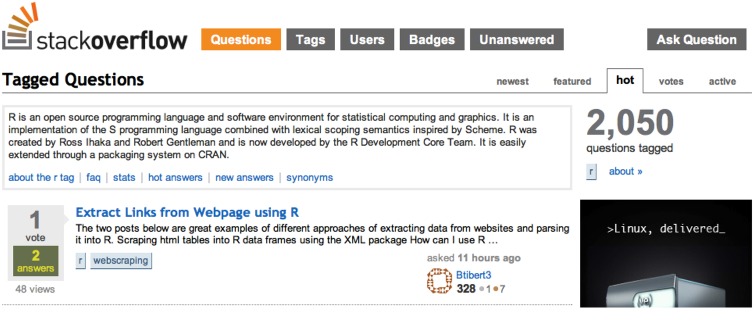
R-Seek Search Engine

`http://www.rseek.org`



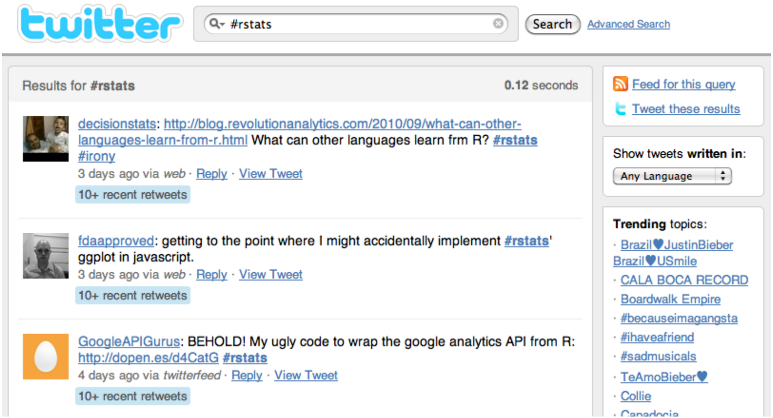
Stackoverflow

http://stackoverflow.com/questions/tagged/r



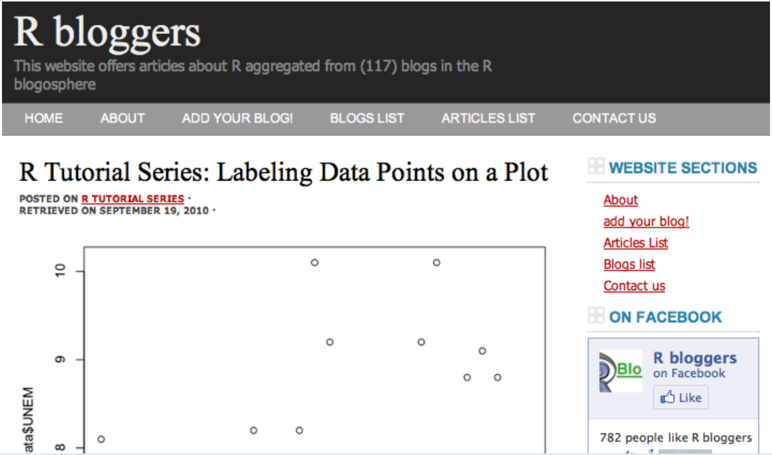
Twitter

`http://search.twitter.com/search?q=#rstats`



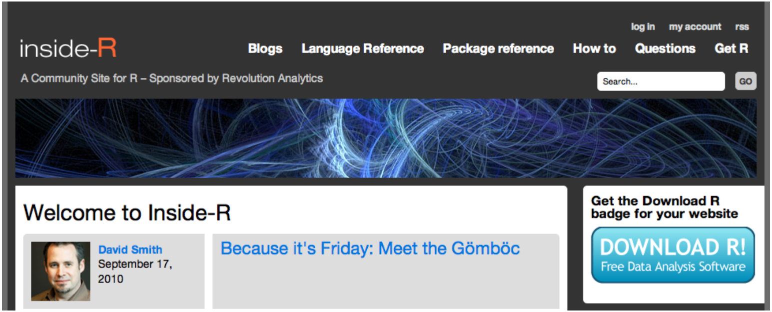
R-bloggers

`http://www.r-bloggers.com/`



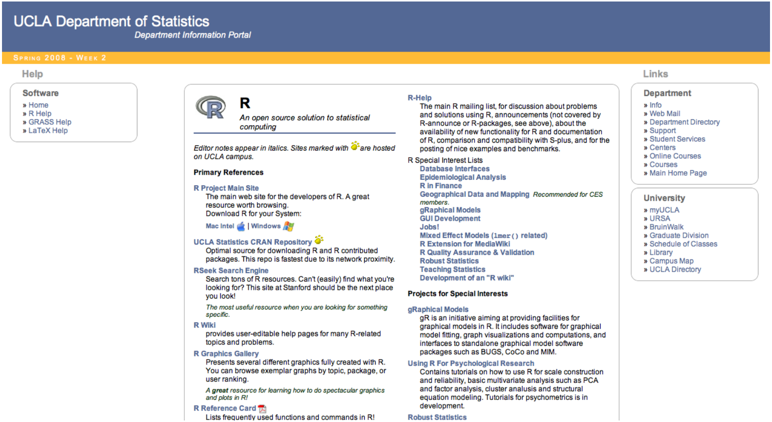
Inside R

`http://www.inside-r.org/`



UCLA Statistics Information Portal

http://info.stat.ucla.edu/grad/



UCLA Department of Statistics
Department Information Portal

Spring 2008 - Week 2

Help

Software
» Home
» R Help
» GRASS Help
» LaTeX Help

R
An open source solution to statistical computing

Editor notes appear in italics. Sites marked with are hosted on UCLA campus.

Primary References

R Project Main Site
The main web site for the developers of R. A great resource worth browsing.
Download R for your System:
Mac Intel | Windows

UCLA Statistics CRAN Repository
Optimal source for downloading R and R contributed packages. This repo is fastest due to its network proximity.

RSeek Search Engine
Search tons of R resources. Can't (easily) find what you're looking for? This site at Stanford should be the next place you look!
The most useful resource when you are looking for something specific.

R Wiki
provides user-editable help pages for many R-related topics and problems.

R Graphics Gallery
Presents several different graphics fully created with R. You can browse exemplar graphs by topic, package, or user ranking.
*A **great** resource for learning how to do spectacular graphics and plots in R!*

R Reference Card
Lists frequently used functions and commands in R!

R-Help
The main R mailing list, for discussion about problems and solutions using R, announcements (not covered by R-announce or R-packages, see above), about the availability of new functionality for R and documentation of R, comparison and compatibility with S-plus, and for the posting of nice examples and benchmarks.

R Special Interest Lists
Database Interfaces
Epidemiological Analysis
R in Finance
Geographical Data and Mapping *Recommended for CES members.*
gRaphical Models
GUI Development
Jobs!
Mixed Effect Models (`lmer()` related)
R Extension for MediaWiki
R Quality Assurance & Validation
Robust Statistics
Teaching Statistics
Development of an "R wiki"

Projects for Special Interests

gRaphical Models
gR is an initiative aiming at providing facilities for graphical models in R. It includes software for graphical model fitting, graph visualizations and computations, and interfaces to standalone graphical model software packages such as BUGS, CoCo and MIM.

Using R For Psychological Research
Contains tutorials on how to use R for scale construction and reliability, basic multivariate analysis such as PCA and factor analysis, cluster analusis and structural equation modeling. Tutorials for psychometrics is in development.

Robust Statistics

Links

Department
» Info
» Web Mail
» Department Directory
» Support
» Student Services
» Centers
» Online Courses
» Courses
» Main Home Page

University
» myUCLA
» URSA
» BruinWalk
» Graduate Division
» Schedule of Classes
» Library
» Campus Map
» UCLA Directory

UCLA Statistical Consulting Center

E-consulting and Walk-in Consulting - `http://scc.stat.ucla.edu`

R : Statistical Consulting Center Forums

Statistical Consulting Center

Forum for the R statistical computing and graphical language and environment.

Goto: Forum List • New Topic • Search • Control Center • Private Messages • Log Out

Current Page: 1 of 1 **Pages:** 1

Subject	Views	Written By	Posted
weights case	161	joscani	03/09/2009 09:01AM
Re: weights case	75	David Diez	03/10/2009 08:35AM
Re: weights case	96	joscani	03/10/2009 11:41AM
Re: weights case	66	Jan de Leeuw	03/11/2009 06:07PM
Adding matrices in an array	190	Mine Cetinkaya	03/02/2009 03:06PM
Re: Adding matrices in an array	85	David Diez	03/02/2009 06:33PM
Re: Adding matrices in an array	54	Mine Cetinkaya	03/03/2009 07:49AM
Re: Adding matrices in an array	53	David Diez	03/03/2009 07:59AM
Re: Adding matrices in an array	55	Mine Cetinkaya	03/03/2009 09:11AM

SCC Mini-Courses

UCLA Department of Statistics

Statistical Consulting Center

Welcome > Mini-courses > Materials from Past Mini-Courses > Winter 2010 Mini-Course Materials

Spring 2010 Mini-Course Materials

Spring 2010 Schedule

Date	*Topic*	*Presenter*	*Materials*
Week 1			
March 29	Organizational Meeting for Statistics 285	Mahtash Esfandiari	
Week 2			
April 5	R Programming & Graphics I: Introduction to Programming and Graphics	Brigid Wilson	
April 7	R Programming II: Data Manipulation and Functions	Denise Ferrari	
Week 3			
April 12	LaTeX I: Writing a Document, Paper, or Thesis	David Diez	
April 14	LaTeX II: Bibliographies, Style and Math	David Diez	
Week 4			
April 19	LaTeX III: Sweave – Embedding R in LaTeX	Colin Rundel	
April 21	LaTeX IV: Academic Talks and Presentations	Mine Cetinkaya	
Week 5			
April 26	R Graphics II: Graphics for Exploratory Data Analysis	Irina Kukuyeva	
April 28	R Graphics III: Customizing Graphics	Ryan Rosario	
Week 6			

Part VI

Exercises

Exercises

- Download the following file to your desktop: http://www.stat.ucla.edu/~crundel/SCC/IntermediateR.data
- Open R and run the following command: *load('~/Desktop/IntermediateR.data')*
- Exercise 1 - There is now a variable called *sudokus* which is a 9x9x50 array. This represents the solutions to 50 different sudoku puzzles, some of which are right some of which are wrong. Using what you've learned today find the indexes of the correct solutions. Hint - there are only 7.
- Exercise 2 - The *scores* data frame used earlier is also in the data file. Expand the data frame such that there is an appropriately named quiz, exam, and overall average for each student as well as an average and standard deviation for each exam and quiz.