R Programming

Lesson 2

Insight campus

Table of Contents

Contents_Useful tricks

Grouped Expressions

Grouped expressions

- Statements can be grouped together using braces '{'and'}'.
- A group of statements is sometimes called a *block*.
- Blocks are not evaluated until a new line is entered after the closing brace.

```
> {
+ a <- 2
+ b <- 1:9
+ a * b
+ }
[1] 2  4  6  8  10  12  14  16  18
```

Writing Functions

Writing your own functions

- The R language allows the user to create objects of mode function. These are true R functions that are stored in a special internal form and may be used in further expressions and so on. In the process, the language gains enormously in power, convenience and elegance, and learning to write useful functions is one of the main ways to make your use of R comfortable and productive.

A function is defined by an assignment of the form

name <- function(arg_1, arg_2, …) expression

- The expression is an R expression, (usually a grouped expression).
- The value of the expression is the value returned for the function.
- A call to the function then usually takes the form *name(expr_1, expr_2, …)*.

Writing your own Functions - Example

Polynomial(quadratic) function

```
> f <- function(x) x^2 + 1
> x <- seq(-1, 1, 0.1)
> y <- f(x)
> plot(x,y, type ="l")
```

odd-even function

```
> odd.even <- function(x){
+   if(x %% 2 == 1) y <- "odd" else y <- "even"
+   y
+ }
> odd.even(10)
[1] "even"
```

Control structures

If

- The **if/else** statement conditionally evaluates two statements. There is a *condition* which is evaluated and if the *value* is **TRUE** then the first statement is evaluated; otherwise the second statement will be evaluated.

- Formal syntax

  if (*condition1*) *statement1*

  if (*condition1*) *statement1* else *statement2*

  if (*condition1*) statement1 else if (*condition2*) statement2 else *statement3*

```
> x <- 75
> if (x >= 90) "A"
> if (x >= 90) "A" else "B"
[1] "B"
> if (x >= 90) "A" else if (x >= 80) "B" else "C"
[1] "C"
```

Example

Median

- The median of a finite list of numbers can be found by arranging all the observations from the lowest value to the highest value and picking the middle one.
- If there is an even number of observations, then there is no single middle value; the median is then usually defined to be the mean of the two middle values.

```
> x <- c(3, 6, 4, 7, 5, 6, 11, 4, 7, 9)
> x.srt <- sort(x)
> x.len <- length(x)
> if (x.len %% 2 == 1) {
+  x.mode <- x.srt[(x.len + 1)/2]
+ } else {x.mode <- ((x.srt[x.len / 2] + x.srt[x.len/2 + 1]) / 2)
+ }
> x.mode
```

Looping

- Looping is the repeated evaluation of a statement or block of statements.
- R has three statements that provide explicit looping.
  - for, while and repeat.
- The two built-in constructs
  - next and break
- Each of the three statements returns the value of the last statement that was evaluated.

Looping

for

- Syntax
  - for (*name* in *vector*) *statement1*
    - vector can be either a vector or a list.
- For each element in *vector* the variable *name* is set to the value of that element and *statement1* is evaluated.

```
> for (i in 10^(0:4)) print (sum(1:i))

> x <- colors()
> for (i in 1:length(x)) {
+ if (i %% 100 == 0)
+ cat("x[", i, "]:", x[i], "\n", sep = "")
+ }
x[100]:darkred
x[200]:gray47
x[300]:grey39
x[400]:lightblue1
x[500]:orange2
x[600]:slategray1
```

Looping

while

- Syntax
  - while (*condition*) *statement1*
- *condition* is evaluated and if its value is **TRUE** then *statement1* is evaluated.
- This process continues until *statement1* evaluates to **FALSE**.

```
> x <- colors()
> i <- 1
> while(i <= length(x)) {
+   if(x[i] == "orange") {
+     cat("x[", i, "]:", x[i], "\n", sep = "")
+     break
+   }
+   i <- i + 1
+ }
x[498]:orange
```

Looping

repeat

- Syntax
  - repeat *statement*

- When using **repeat**, statement must be a block statement.
- You need to both perform some computation and test whether or not to break from the loop and usually this requires two statements.

```
> x <- colors()
> i <- 1
> repeat {
+   if(i > length(x)) break
+   if(x[i] == "orange") {
+     cat("x[", i, "]:", x[i], "\n", sep = "")
+     break
+   }
+   i <- i + 1
+ }
x[498]:orange
```

Example

Factorial

- In mathematics, the factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example,

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

```
> n <- 5; x <- 1
> fac <- function(x) {
+   for(i in n:1) x <- x * i
+   x <- 1; i <- n
+   while(i > 0) {
+     x <- x * i
+     i <- i - 1
+   }
+   x
+ }
> fac(5)
[1] 120
```

Example

CLT

- In probability theory, the **central limit theorem(CLT)** states that, given certain conditions, the arithmetic mean of a sufficiently large number of iterates of independent random variables, each with a well-defined expected value and well-defined variance, will be approximately normally distributed.

```
> n <- 10; p <- 0.9; N <- 30
> inter <- 1000
> x.bar <- rep(NA, inter)
> for(i in 1:inter) x.bar[i] <- mean(rbinom(N, n, p))

> hist(x.bar, prob = T)
> curve(dnorm(x, mean = n*p, sd = sqrt(n*p*(1-p)/N)), add = T, col = "red")
```

Concatenate and Print

cat function

- Outputs the objects, concatenating the representations.
- cat performs much less conversion than print.

```
cat(…, file = “”, sep = “”, fill = FALSE, labels = NULL,
    append = FALSE)
```

Reading data

Command line

- Using objects: vector, matrix, data.frame, etc.

Accessing built-in datasets

- Using function: data()

External files

- Using functions: read.table(), scan(), read.fwf(), etc.

Accessing built-in datasets

data function

- Loads specified data sets, or list the available data sets.

```
data(…, list = character(), package = NULL, lib.loc = NULL,
        verbose = getOption("verbose"), envir = .GlobalEnv)
```

```
> data()                                  # Data sets in package "datasets".
> data(women)
> data("women", package = "datasets")
```

Reading data from files

- Large data objects will usually be read as values from external files.

Functions

- read.table
  - The function read.table has for effect to create a data frame, and so is the main way to read data in tabular form.
- scan
  - The function scan is more flexible than read.table. A difference is that it is possible to specify the mode of the variables.
- read.fwf
  - The function read.fwf can be used to read in a file some data in fixed width format.

Reading data from files

read.table function

- Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

```
read.table(file, header = FALSE, sep = "", quote = "\"",
        dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
        row.names, col.names, as.is = !stringsAsFactors,
        na.strings = "NA", colClasses = NA, nrows = -1,
        skip = 0, check.names = TRUE, fill = !blank.lines.skip,
        strip.white = FALSE, blank.lines.skip = TRUE,
        comment.char = "#",
        allowEscapes = FALSE, flush = FALSE,
        stringsAsFactors = default.stringsAsFactors(),
        fileEncoding = "", encoding = "unknown", text,
        skipNul =  FALSE)
```

Reading data from files - Example

1. Data file has *name* for each variable and *row label*.

- File: “houses.data”

```
> getwd()
> setwd()
> list.files()
[1] "fwf_data.txt" "houses.csv"   "houses.csv2"  "houses.data"  "houses.data2"
"houses.txt"
> HousePrice1 <- read.table(“houses.data")
> HousePrice1
   Price Floor Area Rooms Age Cent.heat
01 52.00   111  830     5 6.2         no
02 54.75   128  710     5 7.5         no
03 57.50   101 1000     5 4.2         no
04 57.50   131  690     6 8.8         no
05 59.75    93  900     5 1.9        yes
```

Reading data from files - Example

2. Data file may omit the *row label* column.

- File: "houses.data2"

```
> HousePrice2 <- read.table("houses.data2", header = T)
> HousePrice2
  Price Floor Area Rooms Age Cent.heat
1 52.00   111  830     5 6.2        no
2 54.75   128  710     5 7.5        no
3 57.50   101 1000     5 4.2        no
4 57.50   131  690     6 8.8        no
5 59.75    93  900     5 1.9       yes
```

Reading data from files - Example

3. Comma-separated values; CSV

- File: “houses.csv”

```
> HousePrice3 <- read.table("houses.csv", header = T, sep = “,”)
> # or
> HousePrice3 <- read.csv("houses.csv")
> HousePrice3
  Price Floor Area Rooms Age Cent.heat
1 52.00   111  830     5 6.2         no
2 54.75   128  710     5 7.5         no
3 57.50   101 1000     5 4.2         no
4 57.50   131  690     6 8.8         no
5 59.75    93  900     5 1.9        yes
```

Reading data from files - Example

4. CSV2

- File: “houses.csv2”

```
> HousePrice4 <- read.table("houses.csv2", header = T, sep = ";", dec = ",")
> # or
> HousePrice4 <- read.csv2("houses.csv2")
> HousePrice4
  Price Floor Area Rooms Age Cent.heat
1 52.00   111  830     5 6.2         no
2 54.75   128  710     5 7.5         no
3 57.50   101 1000     5 4.2         no
4 57.50   131  690     6 8.8         no
5 59.75    93  900     5 1.9        yes
```

Reading data from files - Example

5. Delimited file - defaulting to the TAB character for the delimiter.

- File: “houses.txt”

```
> HousePrice5 <- read.table("houses.txt", header = T, sep = "\t", dec
= ".")
> # or
> HousePrice5 <- read.delim("houses.txt")
> HousePrice5
  Price Floor Area Rooms Age Cent.heat
1 52.00    111  830     5 6.2         no
2 54.75    128  710     5 7.5         no
3 57.50    101 1000     5 4.2         no
4 57.50    131  690     6 8.8         no
5 59.75     93  900     5 1.9        yes
```

Writing data

write.table function

- The function **write.table** writes in a file an object, typically a data frame but this could well be another kind of object.
  - vector, matrix

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = “”)

write.csv(…)

write.csv2(…)
```

Writing data - Example

TAB Delimited file & CSV

```
> data(iris)
> out.data <- iris[1:4]


# TAB character for the delimiter
> write.table(out.data, "iris.txt", quote = F, sep = "\t",
  row.names = F, col.names = T)


# CSV
> write.table(out.data, "iris.csv", quote = T, sep = ",",
  row.names = F, col.names = T)
```

Useful Tricks

See R Cookbook Ch.12

Inserting Data into a Vector

Problem

- You want to insert one or more data items into a vector.

Solution

- Despite its name, the **append** function inserts data into a vector by using the after parameter, which gives the insertion point for the new item or items:

  append(*vector, newvalues,* after = *n*)

Discussion

```
> append(1:10, NA, after = 3)
 [1]  1  2  3 NA  4  5  6  7  8  9 10
```

Combining Multiple Vectors into One Vector and a Factor

Problem

- You have several groups of data, with one vector for each group.

Solution

- Create a list that contains the vectors. Use the **stack** function to combine the list into a two-column data frame:

```
comb <- stack(list(v1 = v1, v2 = v2, v3 = v3))      # Combine 3 vectors
```

Discussion

```
> comb <- stack(list(group1 = 1:2, group2 = 3:4))
> print(comb)
  values   ind
1      1 group1
2      2 group1
3      3 group2
4      4 group2
```

Appending Rows to a Data Frame

Problem

- You want to append one or more new rows to a data frame.

Solution

- Use the **rbind** function to append the temporary data frame to the original data frame.

  rbind(*vectors or matrices*)

Discussion

```
> dt.ori <- data.frame(x = 1:3, y = letters[1:3])
> dt.tmp <- data.frame(x = 4:5, y = letters[4:5])
> dt <- rbind(dt.ori, dt.tmp)
> dt
  x y
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
```

Selecting Rows and Columns More Easily

Problem

- You want an easier way to select rows and columns from a data frame or matrix.

Solution

- Use the **subset** function.

  subset(*x*, *subset*, *select*)

Discussion

```
> data(airquality)
> subset(airquality, Temp > 80, select = c(Ozone, Temp))
> subset(airquality, Day == 1, select = -Temp)
> subset(airquality, select = Ozone:Wind)
```

Removing NAs from a Data Frame

Problem

- Your data frame contains NA values, which is creating problems for you.

Solution

- Use **na.omit** to remove rows that contain any NA values.

  na.omit(*data frame, ...*)

Discussion

```
> df <- data.frame(x = c(NA, 2, 3), y = c(0, 10, NA))
> df.clean <- na.omit(df)
> df.clean
  x  y
2 2 10
```

Combining Two Data Frames

Problem

- You want to combine the contents of two data frames into one data frame.

Solution

- To combine the columns of two data frames side by side, use **cbind:**

  cbind(vectors or matrices)

Combining Two Data Frames

Discussion

```
> dt.ori <- data.frame(x = 1:3, y = letters[1:3]); dt.ori
  x y
1 1 a
2 2 b
3 3 c
> dt <- cbind(dt.ori, z = 0)
> dt
  x y z
1 1 a 0
2 2 b 0
3 3 c 0
```

Combining Two Data Frames

Discussion

```
> rbind(dt.ori, z = 0)                # Check out for the recycling rule
  x    y
1 1    a
2 2    b
3 3    c
z 0 <NA>
경고메시지:
In `[<-.factor`(`*tmp*`, ri, value = 0) :
  invalid factor level, NA generated

> rbind(dt.ori, data.frame(x = 0, y = letters[4]))
  x y
1 1 a
2 2 b
3 3 c
4 0 d
```

Accessing Data Frame Contents More Easily

Problem

- Your data is stored in a data frame. You are getting tired of repeatedly typing the data frame name and want to access the columns more easily.

Solution

- For repetitive access, use the **attach** function to insert the data frame into your search list.

  attach(*data.frame or list*)

  detach()

Accessing Data Frame Contents More Easily

Discussion

```
> search()
 [1] ".GlobalEnv"        "tools:rstudio"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
> summary(women$height)          # refers to a variable 'height' in the data frame
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   58.0    61.5    65.0    65.0    68.5    72.0

> attach(women)
> search()
> summary(height)                # The same variable now available by name

> detach()                       # remove the second location in the search list
> search()
```

Converting One Atomic Value into Another

Problem

- You have a data value which has an atomic data type:
  - character, complex, double, integer, or logical

Solution

- as.character(x)
- as.complex(x)
- as.numeric(x) or as.double(x)
- as.integer(x)
- as.logical(x)

Converting One Structured Data Type into Another

Problem

- You want to convert a variable from one structured data type to another.

Solution

- as.data.frame(x)
- as.list(x)
- as.matrix(x)
- as.vector(x)

Converting One Structured Data Type into Another

Discussion

From	To	How
Vector	List	as.list(vec)
	Matrix	To create a 1-column matrix: cbind(vec) or as.matrix(vec)
		To create a 1-row matrix: rbind(vec)
		To create an n × m matrix: matrix(vec, n, m)
	Data frame	To create a 1-column data frame: as.data.frame(vec)
		To create a 1-row data frame: as.data.frame(rbind(vec))
List	Vector	unlist(lst)
	Matrix	To create a 1-column matrix: as.matrix(lst)
		To create a 1-row matrix: as.matrix(rbind(lst))
		To create an n × m matrix: matrix(lst, n, m)
	Data frame	If the list elements are columns of data: as.data.frame(lst)
		If the list elements are rows of data: see Recipe 5.19 of R Cookbook

Converting One Structured Data Type into Another

Discussion

From	To	How
Matrix	Vector	as.vector(mat)
	List	as.list(mat)
	Data frame	as.data.frame(mat)
Data frame	Vector	To convert a 1-row data frame: dfrm[1,]
	Data frame	To convert a 1-column data frame: dfrm[, 1] or dfrm[[1]]
	List	as.list(dfrm)
	Matrix	as.matrix(dfrm)

Splitting a Vector into Groups

Problem

- You have a vector. Each element belongs to a different group, and the groups are identified by a grouping factor. You want to split the elements into the groups.

Solution

- Suppose the vector is x and the factor is f. You can use the **split** function:

```
groups <- split(x, f)
```

- Alternatively, you can use the **unstack** function:

```
groups <- unstack(data.frame(x, f))
```

Splitting a Vector into Groups

Discussion

```
> library(MASS)
> group <- split(Cars93$MPG.city, Cars93$Origin)
> group
$USA
 [1] 22 19 16 19 16 16 25 25 19 21 18 15 17 17 20 23 20 29 23 22 17 21 18
[24] 29 20 31 23 22 22 24 15 21 18 17 18 23 19 24 23 18 19 23 31 23 19 19
[47] 19 28

$`non-USA`
 [1] 25 18 20 19 22 46 30 24 42 24 29 22 26 20 17 18 18 29 28 26 18 17 20
[24] 19 29 18 29 24 17 21 20 33 25 23 39 32 25 22 18 25 17 21 18 21 20
```

Applying a Function to Every Row

Problem

- You have a matrix. You want to apply a function to every row, calculating the function result for each row.

Solution

- Use the **apply** function. Set the second argument to 1 to indicate row-by-row application of a function:
  results <- apply(*matrix or array*, 1, function, …)

Discussion

```
   # Mauna Loa Atmospheric CO2 Concentration
> data(co2); plot(co2)
> means <- apply(matrix(co2, ncol = 12, byrow = T), 1, mean)
> names(means) <- 1959:1997
> plot(means)
```

Applying a Function to Every Column

Problem

- You have a matrix or data frame, and you want to apply a function to every column.

Solution

- For a matrix, use the **apply** function. Set the second argument to 2, which indicates column-by-column application of the function: row-by-row application of a function:

```
results <- apply(matrix or array, 2, function, …)
```

- For a data frame, use the **lapply or sapply** functions. (Applying a Function to Each List Element)

```
lst <- lapply(vector or list, function, …)          # lapply returns a list
vec <- sapply(vector or list, function, …)
```

Applying a Function to Every Column

Discussion

```
   # For a matrix
> data(co2, package = "datasets")                    # same as data(co2)
> means <- apply(matrix(co2, ncol = 12, byrow = T), 2, mean)
> names(means) <- 1:12
> plot(means)
```

```
   # For a data frame
> data(iris, package = "datasets")                   # same as data(iris)
> apply(iris[1:4], 2, mean)
> sapply(iris[1:4], mean)
> lapply(iris[1:4], mean)
> apply(iris[1:4], 2, summary)
> sapply(iris[1:4], summary)
> lapply(iris[1:4], summary)
```

Applying a Function to Groups of Data

Problem

- Your data elements occur in groups. You want to process the data by groups.

Solution

- Create a grouping factor (of the same length as your vector) that identifies the group of each corresponding datum. Then use the **tapply** function, which will apply a function to each group of data:
  tapply(*vector, list of one or more factors, function, …*)

Applying a Function to Groups of Data

Discussion

```
   # contingency table from data.frame: array with named dimnames
> data("warpbreaks")
> tapply(warpbreaks$breaks, warpbreaks[, -1], sum)
     tension
wool   L   M   H
   A 401 216 221
   B 254 259 169
```

Applying a Function to Groups of Rows

Problem

- You want to apply a function to groups of rows within a data fame.

Solution

- Define a grouping factor—that is, a factor with one level (element) for every row in your data frame—that identifies the data groups.
- For each such group of rows, the **by** function puts the rows into a temporary data frame and calls your function with that argument.
  by(*data frame or matrix, factor or a list of factors, function, …*)

Discussion

```
> by(iris, iris[, 5], summary)
```

Applying a Function to Groups of Rows

Discussion

```
> by(iris, iris[, 5], summary)
iris[, 5]: setosa
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.300   Min.   :1.000   Min.   :0.100   setosa    :50
 1st Qu.:4.800   1st Qu.:3.200   1st Qu.:1.400   1st Qu.:0.200   versicolor: 0
 Median :5.000   Median :3.400   Median :1.500   Median :0.200   virginica : 0
 Mean   :5.006   Mean   :3.428   Mean   :1.462   Mean   :0.246
 3rd Qu.:5.200   3rd Qu.:3.675   3rd Qu.:1.575   3rd Qu.:0.300
 Max.   :5.800   Max.   :4.400   Max.   :1.900   Max.   :0.600
------------------------------------------------------------------------------
iris[, 5]: versicolor
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.900   Min.   :2.000   Min.   :3.00   Min.   :1.000   setosa    : 0
 1st Qu.:5.600   1st Qu.:2.525   1st Qu.:4.00   1st Qu.:1.200   versicolor:50
 Median :5.900   Median :2.800   Median :4.35   Median :1.300   virginica : 0
 Mean   :5.936   Mean   :2.770   Mean   :4.26   Mean   :1.326
 3rd Qu.:6.300   3rd Qu.:3.000   3rd Qu.:4.60   3rd Qu.:1.500
 Max.   :7.000   Max.   :3.400   Max.   :5.10   Max.   :1.800
------------------------------------------------------------------------------
iris[, 5]: virginica
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.900   Min.   :2.200   Min.   :4.500   Min.   :1.400   setosa    : 0
 1st Qu.:6.225   1st Qu.:2.800   1st Qu.:5.100   1st Qu.:1.800   versicolor: 0
 Median :6.500   Median :3.000   Median :5.550   Median :2.000   virginica :50
 Mean   :6.588   Mean   :2.974   Mean   :5.552   Mean   :2.026
 3rd Qu.:6.900   3rd Qu.:3.175   3rd Qu.:5.875   3rd Qu.:2.300
 Max.   :7.900   Max.   :3.800   Max.   :6.900   Max.   :2.500
```

Applying a Function to Parallel Vectors or Lists

Problem

- You want to apply the function element-wise to vectors and obtain a vector result.

Solution

- Use the **mapply** function. It will apply the function f to your arguments element-wise:

mapply(*function, vectors or lists*)

Discussion

```
> mapply(rep, 1:4, 4:1)
> mapply(rep, times = 1:4, x = 4:1)
> mapply(seq, from = 1, to = 1:10)
```

Applying a Function to Parallel Vectors or Lists

Discussion

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4

[1] 2 2 2

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

```
> mapply(rep, times = 1:4, x = 4:1)
[[1]]
[1] 4

[[2]]
[1] 3 3

[[3]]
[1] 2 2 2

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

Applying a Function to Parallel Vectors or Lists

Discussion

```
> mapply(seq, from = 1, to = 1:10)
[[1]]
[1] 1

[[2]]
[1] 1 2

[[3]]
[1] 1 2 3

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

[[5]]
[1] 1 2 3 4 5

[[6]]
[1] 1 2 3 4 5 6

[[7]]
[1] 1 2 3 4 5 6 7

[[8]]
[1] 1 2 3 4 5 6 7 8

[[9]]
[1] 1 2 3 4 5 6 7 8 9

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

Concatenating Strings

Problem

- You want to join together two or more strings into one string.

Solution

- Use the **paste** function.

  paste(*one or more R objects, sep = “”*)

Discussion

```
> paste(1:12)                          # same as as.character(1:12)
[1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12"
> paste("A", 1:6, sep = "")
[1] "A1" "A2" "A3" "A4" "A5" "A6"
> paste("Today is", date())
```

Extracting Substrings

Problem

- You want to extract a portion of a string according to position.

Solution

- Use **substr**(string, start, end) to extract the substring that begins at start and ends at end.

substr(*character vector, start, stop*)

substr(*character vector, start, stop*) <- *value*

Discussion

```
> substr("Statistics", 1, 4)          # Extract first 4 characters
[1] "Stat"
> substr("Statistics", 7, 10)         # Extract last 4 characters
[1] “tics"

> cities <- c("New York, NY", "Los Angeles, CA", "Peoria, IL")
> substr(cities, nchar(cities) - 1, nchar(cities))
[1] "NY" "CA" "IL"
```

Creating a Sequence of Dates

Problem

- You want to create a sequence of dates, such as a sequence of daily, monthly, or annual dates.

Solution

- The **seq** function is a generic function that has aversion for Date objects.

```
seq(from, to, by, length.out = NULL)
seq.Date(from, to, by, length.out = NULL)      # S3 method for class 'Date'
```

Creating a Sequence of Dates

Discussion

```
> s <- as.Date("2016-01-01"); e <- as.Date("2016-02-01")

> seq(from = s, to = e, by = 1)                    # One month of dates
 [1] "2016-01-01" "2016-01-02" "2016-01-03" "2016-01-04" "2016-01-05" "2016-01-06" "2016-01-07"
 [8] "2016-01-08" "2016-01-09" "2016-01-10" "2016-01-11" "2016-01-12" "2016-01-13" "2016-01-14"
[15] "2016-01-15" "2016-01-16" "2016-01-17" "2016-01-18" "2016-01-19" "2016-01-20" "2016-01-21"
[22] "2016-01-22" "2016-01-23" "2016-01-24" "2016-01-25" "2016-01-26" "2016-01-27" "2016-01-28"
[29] "2016-01-29" "2016-01-30" "2016-01-31" "2016-02-01"

> seq.Date(from = s, to = e, by = 1)
 [1] "2016-01-01" "2016-01-02" "2016-01-03" "2016-01-04" "2016-01-05" "2016-01-06" "2016-01-07"
 [8] "2016-01-08" "2016-01-09" "2016-01-10" "2016-01-11" "2016-01-12" "2016-01-13" "2016-01-14"
[15] "2016-01-15" "2016-01-16" "2016-01-17" "2016-01-18" "2016-01-19" "2016-01-20" "2016-01-21"
[22] "2016-01-22" "2016-01-23" "2016-01-24" "2016-01-25" "2016-01-26" "2016-01-27" "2016-01-28"
[29] "2016-01-29" "2016-01-30" "2016-01-31" "2016-02-01"
```

Creating a Sequence of Dates

Discussion

```
> seq(from = s, by = 1, length.out = 7)                    # Dates, one week apart
[1] "2016-01-01" "2016-01-02" "2016-01-03" "2016-01-04" "2016-01-05"
"2016-01-06" "2016-01-07"

> seq(from = s, by = "month", length.out = 7)          # First of the month for one year
[1] "2016-01-01" "2016-02-01" "2016-03-01" "2016-04-01" "2016-05-01"
"2016-06-01" "2016-07-01"

> seq(from = s, by = "3 months", length.out = 7)    # Quarterly dates for one year
[1] "2016-01-01" "2016-04-01" "2016-07-01" "2016-10-01" "2017-01-01"
"2017-04-01" "2017-07-01"
```

Peeking at Your Data

Problem

- You have a lot of data—too much to display at once. Nonetheless, you want to see some of the data.

Solution

- Use the **head** to view the first few data or rows:
  head(*x*, *n* = *6L*,…)
- Use the **tail** to view the last few data or rows:
  tail(*x*, *n* = *6L*,…)

Discussion

```
> head(women)
> head(women, 10)          # Show first 10 rows
> tail(women)
```

Binning Your Data

Problem

- You have a vector, and you want to split the data into groups according to intervals.
- Statisticians call this *binning* your data.

Solution

- Use the **cut** function. It returns a factor whose levels (elements) identify each datum's group:
  cut(*x, breaks, labels = NULL*)

Discussion

```
> x <- rnorm(1000)
> f <- cut(x, breaks = -3:3)
> summary(f)
(-3,-2] (-2,-1]  (-1,0]   (0,1]   (1,2]   (2,3]    NA's
     14     142     340     340     140      22       2
```

Finding the Position of a Particular Value

Problem

- You have a vector. You know a particular value occurs in the contents, and you want to know its position.

Solution

- The **match** function will search a vector for a particular value and return the position:
  match(*x, vector or NULL, …*)

Discussion

```
> match("s", letters)
[1] 19
```

Sorting a Data Frame

Problem

- You have a data frame. You want to sort the contents, using one column as the sort key.

Solution

- Use the **order** function on the sort key, and then rearrange the data frame rows according to that ordering:

```
dfrm <- dfrm[order(dfrm$key), ]
order(..., na.last = TRUE, decreasing = FALSE)
```

Sorting a Data Frame

Discussion

```
> library(MASS)
> names(Cars93)
 [1] "Manufacturer"      "Model"             "Type"              "Min.Price"
 [5] "Price"             "Max.Price"         "MPG.city"          "MPG.highway"
 [9] "AirBags"           "DriveTrain"        "Cylinders"         "EngineSize"
[13] "Horsepower"        "RPM"               "Rev.per.mile"      "Man.trans.avail"
[17] "Fuel.tank.capacity" "Passengers"       "Length"            "Wheelbase"
[21] "Width"             "Turn.circle"       "Rear.seat.room"    "Luggage.room"
[25] "Weight"            "Origin"            "Make"

> Cars93.sub <- subset(Cars93, select = c("Manufacturer", "Model", "Price"))
> attach(Cars93.sub)
The following objects are masked from Cars93.sub (pos = 3):

    Manufacturer, Model, Price
```

Sorting a Data Frame

Discussion

```
> order(Price)
 [1] 31 44 53 39 80 83 73 88 23 84 45 46 32 62 81 74 79 24 33 13 54 64 42 29
40 25 12 68 47 35 60 72 61 14
[35] 27  6 65 21  1 15 34 16 69 55 17 43 75 86 20 85 76 18 26 56 66 30 70 82
89 41 36 90 37 71  7 38 67 87
[69] 92 91  8 77 28 63  9 93 49 78  3 22  5 58 57  2 51 10 50 52  4 19 11 48 59

> Cars93.srt <- Cars93.sub[order(Price), ]
> head(Cars93.srt)
   Manufacturer    Model Price
31         Ford Festiva   7.4
44      Hyundai   Excel   8.0
53        Mazda     323   8.3
39          Geo   Metro   8.4
80       Subaru   Justy   8.4
83       Suzuki   Swift   8.6
```

Sorting a Data Frame

Discussion

```
   # Sorting by two columns
> Cars93.srt <- Cars93.sub[order(Manufacturer, Price),]
> head(Cars93.srt)
  Manufacturer    Model Price
1        Acura Integra  15.9
2        Acura  Legend  33.9
3         Audi      90  29.1
4         Audi     100  37.7
5          BMW    535i  30.0
6        Buick Century  15.7
> detach()
```