Data Objects (Vectors)

Fritz Scholz

March 16, 2015
Vectors

- A **vector** is a sequence of entities of the **same** type, i.e., numerical, integer, character, logic.
- Single values are just vectors of length 1.

```r
> x <- rev(1:20)  # rev() reverses order of 1:20
> str(x)  # gives structural information about x
int [1:20] 20 19 18 17 16 15 14 13 12 11 ...

> z <- seq(1,4,.5)
> z
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
> seq(0,2,.3)
[1] 0.0 0.3 0.6 0.9 1.2 1.5 1.8
> seq(2,0,-.3)
[1] 2.0 1.7 1.4 1.1 0.8 0.5 0.2
> seq(0,2,-.3)
Error in seq.default(0, 2, -0.3) :
  wrong sign in 'by' argument
> seq(2,0,-.3)
[1] 2.0 1.7 1.4 1.1 0.8 0.5 0.2
```
How to Create Vectors

- We saw `1:20` and `seq(1, 4, .5)`.
- By concatenation of values or other vectors, using `c(., ., .)`.

```r
> x1 <- rev(1:5)
> x2 <- 1:4
> y <- c(x1, x2, 5)
> str(y)
  num [1:10] 5 4 3 2 1 1 2 3 4 5

- Note the type becomes `num` because `5` is viewed as numeric.

```r
c(x1, x2, as.integer(5))
```
Character Vectors

- The elements of character vectors can be single characters or strings of characters, enclosed in single or double quotes.

```r
> a <- c('hearts',"A B C","C","Z")
> a
[1] "hearts" "A B C" "C" "Z"
```

- **Special character vectors** `letters` and `LETTERS`

```r
> letters[2:5]
[1] "b" "c" "d" "e"
> LETTERS[c(1,3,25)]
[1] "A" "C" "Y"
```

- Note the extraction via subscript index vectors inside `[ ]`. 
The backslash character acts as start of an escape sequence inside character strings.

⇒ R documentation on backslash

```r
> c("single ’ quote","double " quote",
+ "double \" quote","single \’ quote")
[1] "single ’ quote" "double \" quote"
    "double \" quote" "single ’ quote"
```

# try the following character string
# using octal triples
# to represent ASCII character codes
```r
> "\251 \345 \101 \202 \222 \242 \263"
```

⇒ ASCII character codes
Logic Vectors

- There are two logic values T and F, without quotes, same as TRUE and FALSE.

```r
> Lvec <- c(T,T,F,F,TRUE)
> Lvec
[1]  TRUE  TRUE FALSE FALSE FALSE

- Logic vectors are most often created by logic expressions

```r
> Lvec <- 1:5 < 2.5 # 2.5 is recycled to length 5
> Lvec
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE
> Lvec+1
[1]  2  2  1  1  1
```

- Logic vectors can be interpreted numerically, T ⇔ 1 and F ⇔ 0, in arithmetic expressions.
Testing Object Types

For each object type there is a test function

- is.numeric()
- is.logical()
- is.character()
- is.integer()
- is.function()
- is.na()

```r
> is.logical(Lvec+0)
[1] FALSE
> is.logical(Lvec)
[1] TRUE
> is.function(myfun)
[1] TRUE
> xx <- c(1,2,3,NA)  # NA is a missing value
> is.na(xx)
[1] FALSE FALSE FALSE TRUE
```
Coercing Object Types

When appropriate you can also coerce an object type. This is not about the value but its storage type in memory.

```r
> as.integer(Lvec)
[1] 1 1 0 0 0
> Lvec+1
[1] 2 2 1 1 1
> is.integer(Lvec+1)
[1] FALSE
> z <- as.integer(Lvec+1)
> z
[1] 2 2 1 1 1
> is.integer(z)
[1] TRUE
```
The `rep()` function is useful in creating vector patterns.

```r
> rep(c(0,0,7),times=3)
[1] 0 0 7 0 0 7 0 0 7

> rep(c(0,0,7),each=3)
[1] 0 0 0 0 0 0 7 7 7

> rep(c(0,0,7),length.out=7)
[1] 0 0 7 0 0 7 0
```

In the last case `c(0,0,7)` is recycled a sufficient number of times.
We already saw two examples `letters[2:5]` and `LETTERS[c(1, 3, 25)]`. `letters[c(5)]` and `letters[5]` both work, but `letters[1, 5]` does not.

Using negative indices in extraction means omitting those indexed vector values.

```r
> (1:10)[-c(5, 7)]
[1]  1  2  3  4  6  8  9 10
> 1:10[-c(5, 7)]
[1]  1  2  3  4  5  6  7  8  9 10
# 10[-c(5, 7)] has precedence over : and is 10
# (1:10) forces the precedence of :
```
If \( \mathbf{x} \) is any vector and \( \mathbf{Lx} \) is a logic vector of same length, then \( \mathbf{x}[\mathbf{Lx}] \) extracts all those vector elements from \( \mathbf{x} \), whose position shows \( \text{T or TRUE} \) in the vector \( \mathbf{Lx} \).

- If \( \mathbf{Lx} \) has shorter length than \( \mathbf{x} \) it is recycled.

```r
> x <- 1:10
> Lx <- x>6
> x[Lx] # same as x[x>6]
[1] 7 8 9 10
> (1:21)[3<c(2,4)]
[1] 2 4 6 8 10 12 14 16 18 20
> 3<c(2,4)
[1] FALSE TRUE
> (1:5)[3<c(2,4)]
[1] 2 4
> x[x!=6]
[1] 1 2 3 4 5 7 8 9 10
```

Note the logic operator \(!=\) meaning "not equal".
Changing Selected Vector Values

> x <- 1:10
> x[5] <- 6
> x
 [1] 1 2 3 4 6 6 7 8 9 10

> x[x>5] <- 6
> x
 [1] 1 2 3 4 6 6 6 6 6 6

> x[-4] <- 6
> x
 [1] 6 6 6 4 6 6 6 6 6 6

> x[4] <- 6
> x
 [1] 6 6 6 6 6 6 6 6 6 6
Logic Operators

- \( x == y \) tests equality between \( x \) and \( y \).
- \( x != y \) tests inequality between \( x \) and \( y \).
- \( x > y, \ x < y, \ x >= y, \) and \( x <= y \) test respective types of inequality.
- \( x \& y \) returns \text{TRUE} when both \( x \) and \( y \) are \text{TRUE}, otherwise \text{FALSE} is returned.
- For numeric \( x, \ y \) only 0 counts as \text{FALSE}.
- \( x | y \) returns \text{TRUE} when \( x \) or \( y \) are \text{TRUE}, otherwise \text{FALSE} is returned.
- \( !x \) return the negation of \( x \), when interpreted as logic value.
- All the above operations work in vectorized form, making \( x \) and \( y \) of same length by recycling the shorter vector.

\[
> \ (1:5)[1:5 > 3] \ # \ replacing \ 3 \ by \ c(3,3,3,3,3)
\]
\[
[1] \ 4 \ 5
\]
Recycling in Binary Operations

- Any binary operation
  +, -, /, *, >, <, >=, <=, !=, ==, |, & etc.
  can be used with vectors on each side.
- The shorter vector is recycled to length of longer vector.
- When the longer length is not a clean multiple of the shorter length the operation is still carried out, but with a warning message.

```r
> 2^(1:3) # same as c(2,2,2)^(1:3)
[1] 2 4 8
> (1:3)^2 # same as (1:3)^c(2,2,2)
[1] 1 4 9
```

- The trigonometric and hyperbolic functions,
- Also sqrt, log, exp, abs, see ?log for more.
Some Useful Vector Functions

- `length(x)` gives the length of the vector `x`.
- `sum(x)` gives the sum of all elements in `x`.
- `prod(x)` gives the product of all elements in `x`.
- `min(x)` and `max(x)` give the minimum and maximum of all elements in `x`.
- `cumsum(x)` gives the cumulative sums of all elements in `x`.
- `cummin(x)` and `cummax(x)` give the cumulative minima and maxima of all elements in `x`.
- `diff(x)` gives the differences of adjacent values in `x`. The resulting vector has length `length(x) - 1`.
- `sort(x)` sorts `x`, numeric or character
- `ind <- order(x) \implies x[ind]` is sorted.

Try out these functions and see documentation on them, concerning missing value `NA` behavior.
Numerical Formatting

- \texttt{round(x, k)} rounds \( x \) to \( k \) decimals.
- \texttt{signif(x, k)} shows the \( k \) significant digits of \( x \).
- If in rounding the first dropped digit is 5, rounding is to the nearest even digit.

\begin{verbatim}
> signif(4.45,2)
[1] 4.4
> signif(4.35,2)
[1] 4.4
\end{verbatim}

- \texttt{trunc(x)} rounds \( x \) to nearest integer in the direction of 0.
- \texttt{floor(x)} gives the greatest integer \( \leq x \).
- \texttt{ceiling(x)} gives the smallest integer \( \geq x \).
- All these functions are vectorized.
Problem of Zeros

> .3/.1-3
[1] -4.440892e-16

> 3/1-3
[1] 0

> seq(0,.4,.1)==.3
[1] FALSE FALSE FALSE FALSE FALSE

> .1==.3/3
[1] FALSE

> seq(0,.4,.1)==(0:4)/10
[1] TRUE TRUE TRUE FALSE TRUE

> unique(c(.3,.4-.1,.5-.2,.6-.3,.7-.4))
[1] 0.3 0.3 0.3
Limitations of representing numbers in a computer. It manifests mostly for numbers that are zero, technically. Sometimes the results are surprising and can bite you. Important to mind when testing x == 0. It would result in FALSE when x is 1.224606e-16. Sometimes you get away with such a test, previous example. It can show in unexpected place like in == tests or in unique. Better test abs(x) <= 1e-12 = 10^{-12}
Naming Vectors

Sometimes it is useful to name vectors.

```r
> month.name
[1] "January"  "February"  "March"
[4] "April"     "May"     "June"
[7] "July"      "August"  "September"
[10] "October"   "November" "December"
# a vector of month names, built into R
> month.days <- c(31,28,31,30,31,30,31,
+ 31,30,31,30,31)
> names(month.days) <- month.name
> month.days
 January  February  March  April
  31      28      31     30
 May  June  July  August
  31     30      31     31
 September  October  November  December
  30     31      30     31
```
> month.days["June"]
June
  30
> names(x) <- letters[1:3]
> x
a b c
1 2 3
> x[letters[2:3]]
b c
  2 3
> names(x) <- letters[c(1:2,2)]
> x
a b b
1 2 3
> x["b"]
b
  2
R has many tools for manipulating text data.
Good coverage is given on pages 76-86 of R for Dummies.
We will introduce just a few here.
Often data may be text messages, e.g., maintenance records.
May need to screen such data for occurrence of part numbers.
Important for scheduling airplane maintenance.
Boeing has a statistical group for data and text mining.
Search CRAN packages for "text mining" ⇒ package tm.
Read documentation on paste, strsplit, and grep.
paste and str split

> str0 <- paste(LETTERS[1:3], 1:3, sep=".")
> str0
[1] "A.1" "B.2" "C.3"
> strsplit(str0, ".", fixed=T)
[[1]]
[1] "A" "1"

[[2]]
[1] "B" "2"

[[3]]
[1] "C" "3"

Note the list form of the output. More on lists later.
grep: R’s 108 Shades of Grey

```r
> grey <- colors()[grep("grey", colors())]
> length(grey)
[1] 108

> head(grey)
[1] "darkgrey"   "darkslategrey" "dimgrey"  "grey"
[5] "grey0"      "grey1"

> tail(grey)
[1] "grey98"     "grey99"    "grey100"
[5] "lightgrey"  "lightslategrey" "slategrey"
```
R’s 108 Shades of Grey

rnorm(1e+06)
The factor data type is the most confusing to new users. It seems to be neither numeric nor character or it seems to be both at the same time. It is used to classify certain categorical data aspects:

- M or F (male/female)
- North, East, South, West
- strongly agree, agree, neutral, disagree, strongly disagree
- green, red, blue, yellow, ...
- Treatment A, B, C
Factors by Example

```r
> directions <- c("North","East","South","South")
> dir.factor <- factor(directions)
> dir.factor
[1] North East South South
Levels: East North South
> as.character(dir.factor)
[1] "North" "East" "South" "South"
> as.numeric(dir.factor)
[1] 2 1 3 3 # numbers reflect alphabetical order
> levels(dir.factor)
[1] "East" "North" "South"
> str(dir.factor)
  Factor w/ 3 levels "East","North",..
> str(dir.factor)
  Factor w/ 3 levels "East","North",...
```

The number coding makes storing factors more compact/efficient.
Dates

- Often data come with dates, providing points on a time axis.
- Differences between dates may serve as life lengths.
- Dates can be incremented.

```r
> dx <- as.Date("2012-1-6")
> dx
[1] "2012-01-06"
> dx <- as.Date("2012/1/6")
> dx
[1] "2012-01-06"
> months(dx)
[1] "January"
> weekdays(dx)
[1] "Friday"
> dx+1:3
[1] "2012-01-07" "2012-01-08" "2012-01-09"
```
Dates come in many formats in external data sets. This can be accommodated via the `format` argument in `as.Date()`.

```r
> as.Date("27 Jun 2012", format="%d %b %Y")
[1] "2012-06-27"
> as.Date("27 June 2012", format="%d %B %Y")
[1] "2012-06-27"
> as.Date("27, Aug, 2012", format="%d,%B,%Y")
[1] NA
> as.Date("27, Aug, 2012", format="%d, %B, %Y")
[1] "2012-08-27"
```

Read the documentation on `as.Date` if uncertain.
> apollo <- "July 20, 1969, 20:17:39"
> apollo.fmt <- "%B %d, %Y, %H:%M:%S"
> xct <- as.POSIXct(apollo, format=apollo.fmt)
> xct
> as.numeric(xct)
[1] -14157741

as.POSIXct expresses date/time in seconds since start of 1970.

Sometimes date/time formats in data sets are not consistent.

Hunt for produced NA’s or clean the data via text manipulation.

There is also as.POSIXlt which represents the date/time elements as a list. Read after we have covered lists.
Arithmetic with Date and Time

> xct
> xct + 24*3600
# increment in seconds for as.POSIXct objects.
> as.Date("1969-07-20") + 12
[1] "1969-08-01"
# increment in days for as.Date objects.
> xct.e <- xct + 77781
> xct.e
[1] "1969-07-21 17:54:00 PDT"
> xct.e - xct
Time difference of 21.60583 hours
> xct.e > xct
[1] TRUE
> Sys.time()
# current system time, local to your computer

> system.time(rnorm(1e7))
  user  system elapsed
 3.712   0.068   3.968
# no output beyond timing
# rnorm(1e7) generates 10000000
# standard normal deviates

> system.time(xr <- rnorm(1e7))
  user  system elapsed
 3.708   0.072   4.029
# also produces xr in workspace
> xr[1:3]
[1]  0.03957654  0.61420864 -1.24596152
> x <- numeric(1e6)
> system.time(for(i in 1:1e6) {x[i] <- rnorm(1)})
  user  system elapsed
  3.81    0.00    3.84
> x <- numeric(1e7)
> system.time(for(i in 1:1e7) {x[i] <- rnorm(1)})
  user  system elapsed
 38.19    0.03   38.24
# growth seems linear with number of iterations
# for 1e7 it is 10 times slower than rnorm(1e7)
Slowness of Storage Updates

```r
> y <- NULL
> system.time(for(i in 1:1e4){y <- c(y,rnorm(1))})
   user  system elapsed
   0.17   0.00   0.17
> y <- NULL
> system.time(for(i in 1:1e5){y <- c(y,rnorm(1))})
   user  system elapsed
  12.23   0.05  12.28
> y <- NULL
> system.time(for(i in 1:1e6){y <- c(y,rnorm(1))})
   user  system elapsed
1617.11  163.24 1783.37
# 1e7 iterations may take 178300 seconds
# or 49.5 hours
```