Stat 302
Statistical Software and Its Applications
Data Objects (Vectors)

Fritz Scholz

Department of Statistics, University of Washington

Winter Quarter 2015

January 8, 2015
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
```
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
> str(c(x1, x2, as.integer(5)))
  int [1:10] 5 4 3 2 1 1 2 3 4 5
```
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 (note the subscripting)

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

- Logic vectors are most often created by logic expressions

```r
> Lvec <- 1:5 < 2.5
> Lvec
[1] TRUE TRUE FALSE FALSE FALSE
```

```r
> Lvec+1
[1] 2 2 1 1 1
```

- Logic vectors can be interpreted numerically, `T ⇔ 1` and `F ⇔ 0`
For each object type there is a test function

- `is.numeric()`, `is.logical()`, `is.character()`, `is.integer()`, `is.function()`

```r
> is.logical(Lvec+0)
[1] FALSE
> is.logical(Lvec)
[1] TRUE
> is.function(myfun)
[1] 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
```
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 and is 10
```
If \( x \) is any vector and \( Lx \) is a logic vector of same length, then
\[ x[Lx] \]
extracts all those vector elements from \( x \), whose position shows \( T \) or \( \text{TRUE} \) in the vector \( Lx \).

If \( Lx \) has shorter length than \( x \) it is recycled (with possible warning, when \( \text{length}(x) \neq \text{multiple of length}(Lx) \)).

```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
> x[x!=6]
[1]  1  2  3  4  5  7  8  9 10
```

Note the logic operator \(!=\) meaning "not equal".
> 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
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 `TRUE` when both \( x \) and \( y \) are `TRUE`, otherwise `FALSE` is returned.
  - For numeric \( x \), \( y \) only 0 counts as `FALSE`.
- \( x | y \) returns `TRUE` when \( x \) or \( y \) are `TRUE`, otherwise `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.

```r
> (1:5)[1:5 > 3] # replacing 3 by c(3,3,3,3,3)
[1] 4 5
```
The `which()` function gives the index positions of a logic vector which hold a `TRUE` value.

```r
> which(6:1 > c(3,4))
[1] 1 2 3

# same as
> which(6:1 > c(3,4,3,4,3,4))
[1] 1 2 3

> 6:1
[1] 6 5 4 3 2 1
> c(3,4,3,4,3,4)
[1] 3 4 3 4 3 4
```
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) → x[ind]` is sorted.
- Try out these functions and see documentation on them, concerning missing value `NA` behavior.
Numerical Formatting

- `round(x, k)` rounds $x$ to $k$ decimals.
- `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.

```r
> signif(4.45, 2)
[1] 4.4
> signif(4.35, 2)
[1] 4.4
```

- `trunc(x)` rounds $x$ to nearest integer in the direction of 0.
- `floor(x)` gives the greatest integer $\leq x$.
- `ceiling(x)` gives the smallest integer $\geq x$.
- All these functions are vectorized.
Most arithmetic operations and many functions are vectorized.

Operations involving 2 vectors \( x \) and \( y \) require that the longer vector is a multiple of the shorter one, warning otherwise.

\[
x+y, \ x-y, \ x*y, \ x/y, \ x^y
\]

add, subtract, multiply, divide, exponentiate componentwise.

\[
> 2^{(1:3)} \quad \text{# same as} \quad c(2,2,2)^{(1:3)}
\]
\[
[1] \ 2 \ 4 \ 8
\]

\[
> (1:3)^2 \quad \text{# same as} \quad (1:3)^{c(2,2,2)}
\]
\[
[1] \ 1 \ 4 \ 9
\]

- The trigonometric and hyperbolic functions, 
  try \texttt{?cos} and \texttt{?cosh}.

- Also \texttt{sqrt}, \texttt{log}, \texttt{exp}, \texttt{abs}, see \texttt{?log} for more.
Problem of Zeros

> sin(pi)
[1] 1.224606e-16

> log(5/2)-log(5)+log(2)
[1] 1.110223e-16

> log(5/2)-log(5)+log(2)+log(exp(1))
[1] 1  # no problem here,

> log(5/2)-log(5)+log(2)+log(exp(1))-1 == 0
[1] TRUE

> log(5/2)-log(5)+log(2)+(log(exp(1))-1)
[1] 1.110223e-16
More on Problem of Zeros

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

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

> unique(c(.3,.4-.1,.5-.2,.6-.3,.7-.4))
[1] 0.3 0.3 0.3

> .6-.3 - .7+.4
[1] 5.551115e-17
Machine Representation of Numbers

- 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 \( \text{FALSE} \) 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 \texttt{unique}.
- Better test \( \text{abs}(x) <= 1e-12 = 10^{-12} \)
Sometimes it is useful to name vectors.

> 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
• R has many tools for manipulating text data.
• Good coverage is given on pages 76-86 of R for Dummies.
• We will skip this 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.
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 data aspects

- M or F (male/female)
- North, East, South, West
- strongly agree, agree, neutral, disagree, strongly disagree
- green, red, blue, yellow, ...
> 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","South": 2 1 3 3

The number coding may be the reason for the existence of factors.
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