Biostatistics 615 - Statistical Computing

Lecture 9 Advanced R – Foundations II

Jian Kang

Oct 22, 2015

Data frames

- Creation:

```
df=data.frame(x = 1:3, y=c("a","b","c"), stringAsFactors=FALSE)
```

- Testing and coercion:

```
typedf(df); class(df); is.data.frame(df); as.data.frame(df)
```

- Combining data frames:

```
cbind(df, data.frame(z=3:1)); rbind(df, data.frame(x=10,y="z"))
```

Special columns

- A data frame is a list of vectors, it is possible for a data frame to have a column that is a list

```
> df = data.frame(x=1:3)
> df$y = list(1:2,1:3,1:4)
> df
  x          y
1 1       1, 2
2 2    1, 2, 3
3 3 1, 2, 3, 4

> df = data.frame(x = 1:3, y = list(1:2,1:3,1:4))
Error in data.frame(1:2, 1:3, 1:4, check.names = FALSE, stringsAsFactors = TRUE) :
  arguments imply differing number of rows: 2, 3, 4

 > df = data.frame(x = 1:3, y = I(list(1:2,1:3,1:4)))
> str(df)
'data.frame': 3 obs. of  2 variables:
 $ x: int  1 2 3
 $ y:List of 3
  ..$ : int  1 2
  ..$ : int  1 2 3
  ..$ : int  1 2 3 4
  ..- attr(*, "class")= chr "AsIs"
```

Subsetting

- R's subsetting operators are powerful and fast. It allows you to pull out the pieces that you're interested in.
- The three subsetting operations
- The six types of subsettings
- Important differences in behaviour for different objects (e.g. vectors, lists, factors, matrices, and data frames)
- The use of subsetting in conjunction with assignment

Subsetting - Vectors

```
x = c(1, 2, 3, 4)
```

- Positive intergers `x[c(3, 1)]`
- Negative integers `x[-c(3, 1)]`
- Logical vectors `x[c(TRUE,TRUE,FALSE, FALSE)]`
- Nothing `x[]`
- Zero `x[0]`
- Character vectors

  ```
  y = setNames(x, letters[1:4])
  y[c("a","b")]
  ```

Lists Subsettings

- Subsetting a list works in the same way as subsetting an atomic vector.
- Use '`[`' will always return a list; '`[[`' and '`$`' to pull out the components of the list.

Matrices and arrays

- One vector for each dimension
- One single vector
- A matrix

```
>vals  = outer(1:5, 1:5, FUN="paste", sep=",")
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,] "1,1" "1,2" "1,3" "1,4" "1,5"
[2,] "2,1" "2,2" "2,3" "2,4" "2,5"
[3,] "3,1" "3,2" "3,3" "3,4" "3,5"
[4,] "4,1" "4,2" "4,3" "4,4" "4,5"
[5,] "5,1" "5,2" "5,3" "5,4" "5,5"
> vals[c(1,3),]
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,] "1,1" "1,2" "1,3" "1,4" "1,5"
[2,] "3,1" "3,2" "3,3" "3,4" "3,5"
> vals[c(4, 15)]
[1] "4,1" "5,3"
> select = matrix(ncol = 2, byrow = TRUE, c(1, 1, 3, 1,2, 4))
> vals[select]
[1] "1,1" "3,1" "2,4"
```

Subsetting - Data frames

- Data frames possess the characteristics of both lists and matrices:
  - if you subset with a single vector, they behave like lists
  - if you subset with two vectors, they behave like matrices

```
> df = data.frame(x = 1:3, y = 3:1, z = letters[1:3])
> df[df$x == 2, ]
  x y z
2 2 2 b
> df[c(1,3),]
  x y z
1 1 3 a
3 3 1 c
#like a list
> df[c("x","z")]
> df[,c("x","z")]
  x z
1 1 a
2 2 b
3 3 c
> str(df[,"x"])
 int [1:3] 1 2 3
> str(df["x"])
'data.frame': 3 obs. of  1 variable:
 $ x: int  1 2 3
```

Subsetting operators

- Simplifying and preserving subsetting

	Simplifying	Preserving
Vector	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`
Factor	`x[1:4, drop=T]`	`x[1:4]`
Array	`x[1, ]` or `x[ , 1]`	`x[1,, drop=F]` or `x[ ,1, drop=F]`
Data frame	`x[,1]` or `x[[1]]`	`x[, 1, drop=F]` or `x[1]`

  - Atomic vector: `x=c(a=1,b=2)` remove names: `x[[1]]; #x[1]`
  - List: `y=list(a=1,b=2)` return the object inside, `y[[1]]; #y[1]`
  - Factor: `z = factor(c("a","b"))` drops any unused levels `levels(z[1, drop=TRUE]); #z[[1]]; #z[1]`
  - Matrix or array: `a=matrix(1:4,nrow=2)` if any of dimensions has length 1, drops that dimension `a[1, ]; # a[1, ,drop=F]`
  - Data frame: `df=data.frame(a=1:2,b=1:2)`; Return a vector `df[[1]]; df[, "a"]; #df[, "a"]]; #df[1];`

Subsetting operators

- `$`: a shorthand operator, `x$y` ⇔ `x[["y", exact=FALSE]]`
- `$` does partial matching, `x = list(abc=1); x$a; #x[["a"]]`
- Missing / out of bounds indices (OOB)

Operator	Index	Atomic	List
`[`	OOB	NA	list(NULL)
`[`	NA_real_	NA	list(NULL)
`[`	NULL	x[0]	list(NULL)
`[[`	OOB	Error& Error	
`[[`	NA_real_	Error	NULL
`[[`	NULL	Error	Error

Subsetting and assignment

- All subsetting operators can be combined with assignment to modify selected values of the input vector

```
> x = 1:5
> (x[c(1,2)] = 2:3)
[1] 2 3 3 4 5
> (x[-1] = 4:1)
[1] 2 4 3 2 1
> (x[c(1,1)] = 2:3)
[1] 3 4 3 2 1
> x[c(1,NA)] = c(1,2)
Error in x[c(1, NA)] = c(1, 2) :
  NAs are not allowed in subscripted assignments
# You can combine logical indices with NA
> (x[c(T,F,NA)] = 1)
[1] 1 4 3 1 1
# Conditionally modifying vectors
> df = data.frame(a = c(1,10,NA))
> df$a[df$a < 5] = 0
> df$a
[1]  0 10 NA
```

Subsetting and assignment

- Subsetting with nothing can be usefull

```
> x  = data.frame(abc = rnorm(1000),cdf=rt(1000,df=5))
# remian as a data frame
> x[] = lapply(x,as.integer)
# become a list
> x = lapply(x,as.integer)
```

- Remove component from a list

```
> x  =  list(a=1, b=2)
> x[["b"]] = NULL
> str(x)
List of 1
 $ a: num 1
```

- Add null to a list

```
> y =  list(a=1)
> y[["b"]] = list(NULL)
> str(y)
List of 2
 $ a: num 1
 $ b:List of 1
  ..$ : NULL
```

Subsetting – Applications

- Lookup tables
- Matching and merging by hand
- Random samples / bootstrap
- Ordering
- Expanding aggregated counts
- Removing columns from data frames
- Selecting rows based on a condition
- Boolean algebra versus sets

Lookup table

Character matching provides a powerful way to make lookup table

```
> x = c("m","f","u","f","f")
> lookup = c(m="male",f="female",u="unknown")
> lookup[x]
        m         f         u         f         f
   "male"  "female" "unknown"  "female"  "female"
> unname(lookup[x])
[1] "male"    "female"  "unknown" "female"  "female"
```

Matching and merging by hand

You may have a more complicated lookup table which has multiple columns of information.
Suppose we have a vector of integer grades and a table that describes their properties

```
> grades = c(1,2,2,3,1)
> info = data.frame(grade=3:1, desc=c("Excellent","Good","Poor"),
                                                  fail=c(FALSE,FALSE,TRUE))
> info
  grade      desc  fail
1     3 Excellent FALSE
2     2      Good FALSE
3     1      Poor  TRUE
```

We want to duplicate the info table so that we have a row for each value in `grades`.

Using match()

```
> id = match(grades, info$grade)
> info[id,]
    grade      desc  fail
3       1      Poor  TRUE
2       2      Good FALSE
2.1     2      Good FALSE
1       3 Excellent FALSE
3.1     1      Poor  TRUE
```

`match` returns a vector of the positions of (first) matches of its first argument in its second.

Using rownames

```
> rownames(info) = info$grade
> info[as.character(grades),]
    grade      desc  fail
1       1      Poor  TRUE
2       2      Good FALSE
2.1     2      Good FALSE
3       3 Excellent FALSE
1.1     1      Poor  TRUE
```

Random sample/ bootstrap

You can use integer indices to perform random sampling or bootstrapping of a vector or data frame

`sample()` generates a vector of indices, then subsetting to access the values

```
> df = data.frame(x=rep(1:3,each=2),y=6:1,z=letters[1:6])
> df[sample(nrow(df)),]
  x y z
2 1 5 b
5 3 2 e
3 2 4 c
1 1 6 a
4 2 3 d
6 3 1 f
> df[sample(nrow(df),3),]
  x y z
5 3 2 e
3 2 4 c
6 3 1 f
> df[sample(nrow(df),6,rep=TRUE),]
    x y z
2   1 5 b
6   3 1 f
1   1 6 a
```

Ordering

order() takes a vector as input and returns an integer vector describing how the subsetted vector should be ordered

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

You can change from ascending to descending order using `decreasing=TRUE`.
By default, any missing values will be put at the end of the vector; however, you can remove them with `na.last=NA` or put at the front with `na.last=FALSE`.

Expanding aggregated counts

Sometimes you get a data frame where identical rows have been collapsed into one and a count column has been added.
rep() and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index

```
> df = data.frame(x = c(2,4,1),y=c(9,11,6),n=c(3,5,1))
> rep(1:nrow(df), df$n)
[1] 1 1 1 2 2 2 2 2 3
> df[rep(1:nrow(df),df$n),]
    x  y n
1   2  9 3
1.1 2  9 3
1.2 2  9 3
2   4 11 5
2.1 4 11 5
2.2 4 11 5
2.3 4 11 5
2.4 4 11 5
3   1  6 1
```

Removing columns from data frames

There are two ways to remove columns from a data frame.
You can set individual columns to NULL

```
> df = data.frame(x=1:3, y=3:1, z=letters[1:3])
> df$z = NULL
```

Or you can subset to return only the columns you want: You can set individual columns to NULL

```
> df = data.frame(x=1:3, y=3:1, z=letters[1:3])
> df[c("x","y")]
  x y
1 1 3
2 2 2
3 3 1
```

If you know the columns you don't want, use set operations to work out which columns to keep .

```
> df = data.frame(x=1:3, y=3:1, z=letters[1:3])
> df[setdiff(names(df),"z")]
  x y
1 1 3
2 2 2
3 3 1
```

Selecting rows based on a condition

The logical subsetting is the most commonly used technique for extracting rows out of a data frame

```
> df = data.frame(x=1:3, y=3:1, z=letters[1:3])
> df
  x y z
1 1 3 a
2 2 2 b
3 3 1 c
> df[df$x>2,]
  x y z
3 3 1 c
> df[df$x>=2 | df$y<=2,]
  x y z
2 2 2 b
3 3 1 c
# subset() is a specialized shorthand function for subsetting data frames
> subset(df,z=="a")
  x y z
1 1 3 a
```

Logical operators

```
! x
x & y
x && y
x | y
x || y
xor(x, y)
```

- `!` indicates logical negation (NOT).
- `&` and `&&` indicate logical AND and `|` and `||` indicate logical OR.
  - The shorter form performs elementwise comparisons in much the same way as arithmetic operators.
  - The longer form evaluates left to right examining only the first element of each vector.
  - Evaluation proceeds only until the result is determined.
  - The longer form is appropriate for programming control-flow and typically preferred in if clauses.
- `xor` indicates elementwise exclusive OR.

Boolean algebra verus Sets

Integer subsetting and logical algebra can be equivalent
Using set operations is more effective when:

- You want to find the first TRUE
- You have very few TRUEs and very many FALSEs; a set representation may be faster and require less storage

`which()` allows you to convert a boolean representation to an integer representation. There is no reverse operation in base R

```
> x = sample(10)<4
> which(x)
[1] 2 7 8
> unwhich = function(x,n){
+ out = rep(FALSE,n)
+ out[x] = TRUE
+ return(out)
+ }
> unwhich(which(x),10)
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

Boolean algebra verus Sets

Let create two logical vectors and their integer equivalents a

```
> (x1 = 1:10 %% 2 == 0)
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
> (x2 = which(x1))
[1]  2  4  6  8 10
> (y1 = 1:10 %% 5 == 0)
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
> (y2 = which(y1))
[1]  5 10
```

Boolean algebra verus Sets

Explore the relationship between `boolean` and `set` operations

```
> x1 & y1
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
> intersect(x2,y2)
[1] 10
> x1 | y1
 [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
> union(x2,y2)
[1]  2  4  6  8 10  5
> x1 & !y1
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
> setdiff(x2,y2)
[1] 2 4 6 8
> xor(x1,y1)
 [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
> setdiff(union(x2,y2),intersect(x2,y2))
[1] 2 4 6 8 5
```