

Statistics 360: Advanced R for Data Science

Lecture 2

Brad McNeney

R Data Structures

Vectors

Matrices and data frames

Logical and relational operators

Aside: Special values

Reading

- ▶ Text, chapters 3 and 4

R Data Structures

R Data Structures

- ▶ Fundamentally, all common data structures in R are vectors, which can be “atomic” or “list”.
- ▶ R has no true scalars; e.g., in `x<-1`, `x` is a vector of length one.
- ▶ Use `str()` to see the structure of an object

Types of objects

- ▶ All R objects have a “type”, that describes how it is stored in computer memory.
- ▶ Common types we will encounter are “logical”, “integer”, “double”, “character” and “list”.
 - ▶ Find the type of an object with `typeof()`.

```
x <- 6 # stores as double by default  
typeof(x)
```

```
## [1] "double"
```

```
y <- 6L # The "L" suffix forces storage as integer  
typeof(y)
```

```
## [1] "integer"
```

Type *versus* Mode

- ▶ In addition to the type of an object, there is its “mode”.
- ▶ The mode of an object is generally the same as its type, but the modes are coarser.
 - ▶ For example, integer and double types are both of mode “numeric”.
- ▶ I don’t understand the need for mode.
 - ▶ The only reason I mention it is that the `str()` function sometimes reports the mode of an object, rather than its type, so we will frequently see reports of numeric objects.

```
mode(x)
```

```
## [1] "numeric"
```

```
mode(y)
```

```
## [1] "numeric"
```

Vectors

Vectors

- ▶ Vectors can be either atomic or list
 - ▶ The elements of an atomic vector must be the same type.
 - ▶ Lists can be comprised of multiple data types
- ▶ Empty vectors can be created by the `vector()` function:

```
# help("vector")  
avec <- vector(mode="numeric",length=4)  
lvec <- vector(mode="list",length=4)
```

- ▶ Data vectors can be created with `c()` or `list()`:

```
avec <- c(54,210,77)  
lvec <- list(54,210,77,c("grey","thin"))
```

Combining vectors

- Use `c()` to combine vectors

```
c(avec, c(100, 101))
```

```
## [1] 54 210 77 100 101
```

```
c(lvec, TRUE)
```

```
## [[1]]
```

```
## [1] 54
```

```
##
```

```
## [[2]]
```

```
## [1] 210
```

```
##
```

```
## [[3]]
```

```
## [1] 77
```

```
##
```

```
## [[4]]
```

```
## [1] "grey" "thin"
```

```
##
```

```
## [[5]]
```

```
## [1] TRUE
```

Vector attributes

- ▶ Vectors have a type and length and, optionally, attributes such as names.
 - ▶ As we have seen, we find the type of an object with `typeof()`.
 - ▶ Find the length of a vector with `length()`.

```
typeof(avec)
```

```
## [1] "double"
```

```
length(avec)
```

```
## [1] 3
```

```
str(avec)
```

```
##  num [1:3] 54 210 77
```

```
typeof(lvec)
```

```
## [1] "list"
```

```
length(lvec)
```

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

```
names(lvec) = c("age", "weight", "height", "hair")  
str(lvec)
```

```
## List of 4
```

```
## $ age : num 54
```

```
## $ weight: num 210
```

```
## $ height: num 77
```

```
## $ hair : chr [1:2] "grey" "thin"
```

► We can specify element names when creating a vector; e.g.:

```
lvec <- list(age=54,weight=210,height=77,hair=c("grey","thin"))
```

Factors

- ▶ The statistical concept of a factor is important in experimental design.
- ▶ Factors are implemented in R as atomic vectors with attributes `class` and `levels`:

```
trt <- factor(c("drug1","placebo","placebo","drug2"))  
attributes(trt)
```

```
## $levels  
## [1] "drug1"    "drug2"    "placebo"  
##  
## $class  
## [1] "factor"
```

```
str(trt)
```

```
## Factor w/ 3 levels "drug1","drug2",...: 1 3 3 2
```

- ▶ The levels are coded numerically (1, 2 and 3) with assigned labels ordered alphabetically ("drug1", "drug2" and "placebo") by default.
- ▶ You can specify an order to the factors with the `level` argument:

```
trt <- factor(c("drug1", "placebo", "placebo", "drug2"),  
              levels=c("placebo", "drug1", "drug2"))  
trt
```

```
## [1] drug1  placebo placebo drug2  
## Levels: placebo drug1 drug2
```

More on object class

- ▶ You can create your own class for an object.
- ▶ Such “meta-data” can be used to tell R how to handle the object; e.g., how to print it, summarize it, etc.

```
class(lvec) <- "prof"  
lvec
```

```
## $age  
## [1] 54  
##  
## $weight  
## [1] 210  
##  
## $height  
## [1] 77  
##  
## $hair  
## [1] "grey" "thin"  
##  
## attr("class")  
## [1] "prof"
```

Subsetting vectors and extracting elements

- ▶ Subset with `[]` or by name:

```
lvec[c(1,3)] # same as lvec[c("age", "height")]
```

```
## $age  
## [1] 54  
##  
## $height  
## [1] 77
```

- ▶ Extract individual elements with `[[`, or `$` for named objects:

```
lvec[[4]]
```

```
## [1] "grey" "thin"
```

```
lvec$hair
```

```
## [1] "grey" "thin"
```


Subsetting factors

- ▶ Subsetting may remove all instances of a level, but the level will be retained in the data structure

```
trt[1:3]
```

```
## [1] drug1    placebo placebo  
## Levels: placebo drug1 drug2
```

- ▶ If subsetting is intended to remove a level of the factor, use `drop=TRUE`

```
trt[1:3,drop=TRUE]
```

```
## [1] drug1    placebo placebo  
## Levels: placebo drug1
```

Subsetting and assignment

- ▶ You can combine subsetting and assignment to change the value of vectors.

```
avec
```

```
## [1] 54 210 77
```

```
avec[2] <- 210  
avec
```

```
## [1] 54 210 77
```

Assignment to vector elements

- ▶ To assign to a vector element, it is clearer to use `[[` rather than `[`.
- ▶ Also, for lists, assignment with `[` requires that the replacement element be of length 1; `[[` does not have this restriction

```
lvec[3:4] <- c("Hi", "there")  
lvec[3:4]
```

```
## $height  
## [1] "Hi"  
##  
## $hair  
## [1] "there"
```

```
lvec[4] <- c("All", "of", "this")
```

```
## Warning in lvec[4] <- c("All", "of", "this"): number of items to replace is  
## a multiple of replacement length
```

```
lvec[4] # Only used first element of replacement vector
```

```
## $hair  
## [1] "All"
```

```
lvec[[4]] <- c("All", "of", "this")  
lvec[3:4]
```

```
## $height  
## [1] "Hi"  
##  
## $hair  
## [1] "All" "of" "this"
```

Coercion: atomic vectors to lists

- ▶ Atomic vectors can be coerced to lists with `as.list()`:

```
avec = c(age=54,weight=210,height=77)
avec
```

```
##      age weight height
##      54     210      77
```

```
as.list(avec)
```

```
## $age
## [1] 54
##
## $weight
## [1] 210
##
## $height
## [1] 77
```

Coercion: lists to atomic vectors

- ▶ Lists can be “flattened” into atomic vectors with `unlist()`:

```
unlist(lvec)
```

```
##      age weight height  hair1  hair2  hair3  
##    "54"  "210"   "Hi"  "All"   "of" "this"
```

- ▶ Notice how the numeric values are coerced to the more flexible character type.
- ▶ The order of flexibility, from least to most, is logical, integer, numeric, character.

Coercion: factors to atomic vectors

- ▶ We saw how to use `factor()` to coerce an atomic vector to a factor.
- ▶ Use `as.vector()` to coerce a factor back to an atomic vector.
- ▶ The result is a character vector. You may need to use `as.numeric()` to coerce to numeric, if required.

```
a <-factor(c(2,1,1,2))  
as.vector(a)
```

```
## [1] "2" "1" "1" "2"
```

```
as.numeric(as.vector(a))
```

```
## [1] 2 1 1 2
```

Matrices and data frames

Matrices and data frames

- ▶ Matrices are implemented as atomic vectors with a “dim” attribute of length 2 (number of rows, number of columns).
- ▶ As an atomic vector, elements of a matrix must all be of the same type.
- ▶ Data frames are lists where each list element has the same length. Thus data frames can include columns of varying type.

Matrices

- ▶ Matrices can be created with the `matrix()` function as in

```
A <- matrix(1:4,nrow=2,ncol=2)
```

```
A
```

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

```
## [1,]    1    3
```

```
## [2,]    2    4
```

- ▶ Here `1:4` is the same as `c(1,2,3,4)`

- ▶ The default is to read the data vector into the matrix column-by-column. To read row-by-row instead use the `byrow=TRUE` argument:

```
A <- matrix(1:4,nrow=2,ncol=2,byrow=TRUE)
A
```

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

Combining matrices

- Combine matrices with `rbind()` and `cbind()`:

```
rbind(A,matrix(c(5,6),nrow=1,ncol=2))
```

```
##      [,1] [,2]  
## [1,]    1    2  
## [2,]    3    4  
## [3,]    5    6
```

```
cbind(A,A)
```

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

Matrix attributes

- ▶ Matrices have a type, dimension and optional attributes such as dimnames (row and column names).

```
typeof(A)
```

```
## [1] "integer"
```

```
dim(A)
```

```
## [1] 2 2
```

```
colnames(A) <- c("var1","var2")
rownames(A) <- c("subj1","subj2")
A
```

```
##      var1 var2
## subj1    1    2
## subj2    3    4
```

```
str(A)
```

```
## int [1:2, 1:2] 1 3 2 4
## - attr(*, "dimnames")=List of 2
## ..$ : chr [1:2] "subj1" "subj2"
## ..$ : chr [1:2] "var1" "var2"
```

Subsetting matrices

- ▶ Subset with `[]` and a comma to separate rows from columns:

```
A[1,1]
```

```
## [1] 1
```

```
A[1,]
```

```
## var1 var2
```

```
##    1    2
```

```
A[,1]
```

```
## subj1 subj2
```

```
##    1    3
```

- ▶ When a subsetting operation leads to a vector, the dimension of the object is “dropped” from 2 to 1. To prevent this use `drop=FALSE`:

```
A[1,,drop=FALSE]
```

```
##          var1 var2
```

```
## subj1      1    2
```

Extracting elements from matrices

- ▶ Can use `[[` to extract elements, but this is not necessary because of the way subsetting to a single element drops to a vector of length 1 by default:

```
A[[1,1]]
```

```
## [1] 1
```

```
A[1,1]
```

```
## [1] 1
```


Coercion: Matrices to/from vectors

- ▶ We have already seen how `matrix()` coerces a vector to a matrix
- ▶ `as.vector()` applied to a matrix removes the `dim` attribute and creates a vector by concatenating columns:

```
as.vector(A)
```

```
## [1] 1 3 2 4
```

Data frames

- ▶ Data frames (class `data.frame`) are the usual way to store data in R.
 - ▶ Rows are intended to be observational units, columns variables
 - ▶ Implemented as a list (columns are list elements), but also behave like a matrix in terms of combining and subsetting.
- ▶ Create with `data.frame`:

```
set.seed(1)
n <- 4
x <- 1:n; y <- rnorm(n,mean=x,sd=1) # multiple commands separated by ;
dd <- data.frame(x=x,y=y) # like making a list
str(dd)
```

```
## 'data.frame':    4 obs. of  2 variables:
## $ x: int  1 2 3 4
## $ y: num  0.374 2.184 2.164 5.595
```

Subsetting and combining data frames like a list

```
dd$x
```

```
## [1] 1 2 3 4
```

```
dd[[1]]
```

```
## [1] 1 2 3 4
```

Subsetting and combining data frames like a matrix

```
dd[1:2,]
```

```
##      x      y
## 1 1 0.3735462
## 2 2 2.1836433
```

```
zz = data.frame(z=runif(4))
cbind(dd,zz)
```

```
##      x      y      z
## 1 1 0.3735462 0.62911404
## 2 2 2.1836433 0.06178627
## 3 3 2.1643714 0.20597457
## 4 4 5.5952808 0.17655675
```

Logical and relational operators

Logical operators

- ▶ The basic logical operators are described in `help("Logic")`.
- ▶ `!` is NOT
- ▶ `&` and `&&` are AND, with `&` acting vector-wise and `&&` acting on scalars
- ▶ `|` and `||` are OR, with `|` acting vector-wise and `||` acting on scalars
- ▶ Make sure you understand the following:

```
x <- c(TRUE,TRUE,FALSE); y <- c(FALSE,TRUE,TRUE)
!x ; x&y ; x&&y ; x|y ; x||y
```

```
## [1] FALSE FALSE TRUE
```

```
## [1] FALSE TRUE FALSE
```

```
## [1] FALSE
```

```
## [1] TRUE TRUE TRUE
```

```
## [1] TRUE
```

- ▶ Notice how `&&` and `||` act on the first element of the vectors `x` and `y` and ignore all the rest.

Relational operators

- ▶ Relational operators can be used to compare values in atomic vectors
 - ▶ See `help("Comparison")`
- ▶ `>` is greater than, `>=` is greater than or equal
- ▶ `<` is less than, `<=` is less than or equal
- ▶ `==` is equal and `!=` is not equal
- ▶ Make sure you understand the following:

```
x <- 1:3; y <- 3:1  
x>y ; x>=y ; x<y ; x<=y ; x==y ; x!=y
```

```
## [1] FALSE FALSE TRUE
```

```
## [1] FALSE TRUE TRUE
```

```
## [1] TRUE FALSE FALSE
```

```
## [1] TRUE TRUE FALSE
```

```
## [1] FALSE TRUE FALSE
```

```
## [1] TRUE FALSE TRUE
```


Subsetting vectors with logical expressions

- Can subset with logicals and [:

```
avec
```

```
##    age weight height  
##    54    210     77
```

```
avec>100
```

```
##    age weight height  
## FALSE    TRUE  FALSE
```

```
avec[avec>100]
```

```
## weight  
##    210
```

```
avec[avec>54 & avec<100]
```

```
## height  
##     77
```

Subsetting matrices with logical expressions

- ▶ Can also subset matrices, but results may not be as expected:

```
A
```

```
##           var1 var2
## subj1      1    2
## subj2      3    4
```

```
A>1
```

```
##           var1 var2
## subj1 FALSE TRUE
## subj2  TRUE TRUE
```

```
A[A>1] # coerces to a vector
```

```
## [1] 3 2 4
```

Subset and assign with logical expressions

- ▶ Combine subset and assign to change the value of objects

```
A[A>1] <- 9
```

```
A
```

```
##      var1 var2
```

```
## subj1    1    9
```

```
## subj2    9    9
```

- ▶ In the above substitution, the vector 9 is shorter than the three elements in A>1 so R “recycles” the 9 three times.

Be careful about recycling:

```
A[A>1] <- c(-10,10) # Throws a warning
```

```
## Warning in A[A > 1] <- c(-10, 10): number of items to replace is not  
## of replacement length
```

```
A # R used c(-10,10), then just the -10
```

```
##      var1 var2  
## subj1    1   10  
## subj2  -10  -10
```

Aside: Special values

Missing values

- ▶ R has a special data code for missing data: NA
- ▶ Test for and set missing values with `is.na()`

```
avec
```

```
##      age weight height  
##      54     210      77
```

```
is.na(avec)
```

```
##      age weight height  
## FALSE  FALSE  FALSE
```

```
is.na(avec) <- 2
```

```
avec
```

```
##      age weight height  
##      54      NA      77
```

Infinite and undefined values

- ▶ R has a special codes for infinite values (Inf) and undefined values (NaN).
- ▶ Test for Inf and NaN with `is.infinite()` and `is.nan()`.

```
ii <- 1/0 ; nn <- 0/0
```

```
ii
```

```
## [1] Inf
```

```
is.infinite(ii)
```

```
## [1] TRUE
```

```
nn
```

```
## [1] NaN
```

```
is.nan(nn)
```

```
## [1] TRUE
```

The null object

- ▶ The null object, NULL, is an un-typed no-value object.
 - ▶ Test for NULL with `is.null()`
 - ▶ NULL can be used to initialize objects that will be created through combining, rbinding, etc.

```
x <- NULL; is.null(x)
```

```
## [1] TRUE
```

```
x <- c(x,1); x <- c(x,2); x
```

```
## [1] 1 2
```

```
# etc., or as a loop (more on these later)
```

```
x <- NULL
for(i in 1:2) {
  x <- c(x,i)
}
x
```

```
## [1] 1 2
```