Statistics 360: Advanced R for Data Science

Lecture 2

Becky Lin

R Data Structures

Vectors

Matrices, data frames and tibbles

Logical and relational operators

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

- Two types of vector: can be either atomic or list
  - **Atomic**: all elements must be the same type.
  - **List** ("generic vectors"): objects can be of different types.
  - (low-key third type) `NULL`: absence of vector.

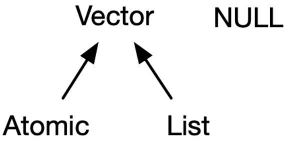


Figure 1: vector types

- R has no true **scalars** (a special syntax to create an individual value), everything that looks like a scalar is actually a vector of one. e.g., `x<-1`.
- Empty vectors can be created by the `vector()` function:

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
```

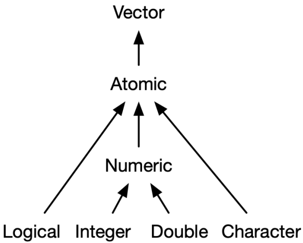
```
## [1] 1 2
```

Types of atomic vector

Four primary types of atomic vectors:

1. Logical: either be written in full (`TRUE` or `FALSE`), or abbreviated (`T`,`F`)
2. Integer: exactly an integer. Assign them by adding `L` behind it (for "long integer"). e.g., -1L, 0L, 1L, 2L, etc.
3. Double: decimal numbers. e.g., 1, 1.0, 1.02 ... Note that `Inf`, `-Inf`, `NaN` are also doubles.
4. Character: anything in quotes.

Integers and doubles are together called "numeric"



Creating empty vectors and Combining vectors

- Use `vector()` to create an empty vector

```
avec <- vector(mode="double",length=4) # shorthand: avec <- double()
lvec <- vector(mode="list",length=3)  # shorthand: levc <- list()
```

- Use `c()` to combine vectors

```
avec <- c(54,210,77)
lvec <- list(54,210,c("grey","thin"))
c(avec,c(100,101))
```

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

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

```
## [[1]]
## [1] 54
##
## [[2]]
## [1] 210
##
## [[3]]
## [1] "grey" "thin"
##
## [[4]]
## [1] TRUE
```

Combining lists

- c() concatenates two lists, which may or may not be what you intend.

```
ll1 <- list(a=1:2,b=3:4); ll2 <- list(a=5:6,b=7:8)
c(ll1,ll2) # not the same as list(ll1,ll2)
```

```
## $a
## [1] 1 2
##
## $b
## [1] 3 4
##
## $a
## [1] 5 6
##
## $b
## [1] 7 8
```

- Use unlist() to remove the list structure.

```
unlist(ll1)
```

```
## a1 a2 b1 b2
##  1  2  3  4
```

- A list of lists:

```
c(list(ll1),list(ll2)) # keep this in mind for the project
```

```
## [[1]]
## [[1]]$a
## [1] 1 2
##
## [[1]]$b
## [1] 3 4
##
##
## [[2]]
## [[2]]$a
## [1] 5 6
##
## [[2]]$b
## [1] 7 8
```

Determine type and length of a vector

- You can determine the type of a vector with `typeof()`
- The special values (infinite and undefined values): `Inf`,`-Inf`, and `NaN` are doubles. Test for `Inf` and `NaN` with `is.infinite()` and `is.nan()`.
- You can check the specific type with `is.logical()`,`is.integer()`,`is.double()`, and `is.character()`.
- Missing values are represented by `NA` which is technically a logical value.
  - This rarely matters because logicals get coerced to other types when needed.
  - Never use == when testing for missingness. It will return `NA` since it is always unkown if two unknowns are equal.
  - Test for and set missing values with `is.na()`.
- Attempting to combine vectors of different types coerces them to the same type. The order of preference is character>integer>double>logical.

```r
typeof(list(a=2,b=3))
```

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

```r
x <- 1:3; typeof(x)
```

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

```r
length(x)
```

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

```r
is.integer(x)
```

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

```r
typeof(NA)
```

```
## [1] "logical"
```

```r
typeof(c(1L, NA)) # logicals get coerced to other types
```

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

```r
typeof(c(Inf,-Inf, NaN))
```

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

```r
typeof(c(2L,TRUE, 98.0)) # different types coerces to the same type
```

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

Test your knowledge

Predict R output for the following R code

```
x <- c(1,2,3, NA)
typeof(x)
is.na(c(1,2,3, NA))
is.na(x) <- 2 # set 2nd element of x to be NA
x
is.finite(c(1,Inf,-Inf, NaN))
is.infinite(c(1,Inf,-Inf, NaN))
is.nan(c(1,Inf,-Inf, NaN))
typeof(c("hello",9L, TRUE))
# exercise from Advanced R
c(1, FALSE)
c("a", 1)
c(TRUE, 1L)
1 == "1" # TRUE
-1 < FALSE # TRUE
"one" < 2 # FALSE
```

Vector attributes

- Attributes are meta information applied to atomic vectors.
- Many common objects (like matrices, arrays, factors, date-times) are just atomic vectors with special attributes.
- Vectors have type,length and, optionally, attributes such as names.
- You get and set attributes with `attr()`.
- You could see all attributes of a vector with `attributes()`.
- Attributes are name-value pairs, and all of the attributes are associated with an object. Below, the vector c(1,2,3) points to two attributes x and y that each have their own values.

```
a <- 1:3
attr(a,"x") <- "abecdf"
attr(a, "y") <- 4:6
attributes(a) # display all attributes
```

```
## $x
## [1] "abecdf"
##
## $y
## [1] 4 5 6
```

```
# attr(a, "x") # retrieve the x attributes of vector a
```

- Set many attributes at the same time with `structure()`

```
b <- structure(1:3, x = "abcdef", y=4:6)
attributes(b)
```

```
## $x
## [1] "abcdef"
##
## $y
## [1] 4 5 6
```

```
b # after operation
```

```
## [1] 1 2 3
## attr(,"x")
## [1] "abcdef"
## attr(,"y")
## [1] 4 5 6
```

- Most attributes are typically lost by most operations except **names** and **dim**

```
attributes(b[[1]])
```

```
## NULL
```

```
attributes(sum(b))
```

```
## NULL
```

Vector attribute - names

- Names: a character vector giving each element of the vector a name. Each name corresponds to a single element.
- You could set names using `attr()`, but you should not.

```
#This method is not recommended
x <- 1:3
attr(x, "names") <- c("a","b","c")
x
```

```
## a b c
## 1 2 3
```

```
attributes(x)
```

```
## $names
## [1] "a" "b" "c"
```

```
names(x)
```

```
## [1] "a" "b" "c"
```

- Names are so special, that you could specify element names when creating a vector.

```r
x <- c(a=1, b=2, c=3)
x
```

```
## a b c
## 1 2 3
```

```r
attributes(x)
```

```
## $names
## [1] "a" "b" "c"
```

```r
names(x)
```

```
## [1] "a" "b" "c"
```

```r
# alternatively
x <- 1:3
names(x) <- c("a","b","c")
x
```

```
## a b c
## 1 2 3
```

```r
x+2:4 # names attribute is preserved
```

```
## a b c
## 3 5 7
```

- Names stay with single bracket subsetting (not double bracket subsetting).

```
names(x[1])
```

```
## [1] "a"
```

```
names(x[1:2])
```

```
## [1] "a" "b"
```

```
names(x[[1]])
```

```
## NULL
```

- Names can be used for subsetting (more in chapter 4)

```
x[["a"]]
```

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

- we can remove names with `unname()`

```
unname(x)
```

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

- Attributes tend to be dropped after operation. If not, they may not make sense.

```
x <- structure(1:3,at1="abc")
y <- structure(3:1,at1="def")
x
```

```
## [1] 1 2 3
## attr(,"at1")
## [1] "abc"
```

```
y
```

```
## [1] 3 2 1
## attr(,"at1")
## [1] "def"
```

```
x[1:2]
```

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

```
x*y
```

```
## [1] 3 4 3
## attr(,"at1")
## [1] "abc"
```

Vector attribute - dimension

- the `dim` attribute makes a vector into a **matrix** (a rectangle of numbers) or an **array** (a block of numbers).
- again, you could use `attr()` to set `dim()`, but you should not.

```
x <- 1:6
attr(x,'dim') <- c(2,3)
x
```

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

```
y <- 1:12
attr(y,'dim') <- c(2,2,3)
y
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
```

- you should either use `matrix()` or `array()` to create these objects, or set the dimension with `dim()`

```
x <- 1:6
dim(x) <- c(2,3)
x
```

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

```
dim(x)
```

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

```
x <- matrix(1:6,nrow=2,ncol=3, byrow=FALSE) # byrow=FALSE,
y <- 1:12
attr(y,'dim') <- c(2,2,3)
y
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
```

- Many of the functions for working with vectors have generalisations for matrices and arrays:

Vector	Matrix	Array
`names()`	`rownames()`, `colnames()`	`dimnames()`
`length()`	`nrow()`, `ncol()`	`dim()`
`c()`	`rbind()`, `cbind()`	`abind::abind()`
—	`t()`	`aperm()`
`is.null(dim(x))`	`is.matrix()`	`is.array()`

Figure 3: Functions for matrices and arrays

- Instead of having `names`, arrays and matrices of `dimnames`. The dimnames of a matrix or an array is a list the same length as the number of dimensions of the array.

```
x <- matrix(1:12,nrow=3,ncol=4)
dimnames(x) = list( c("r1","r2","r3"), c("c1","c2","c3","c4"))
x <- array(1:12, dim = c(2, 2, 3))
dimnames(x) <- list(first = c("a", "b"),
                    second = c("c", "d"),
                    third = c("e", "f", "g"))
dimnames(x)
```

```
## $first
## [1] "a" "b"
##
## $second
## [1] "c" "d"
##
## $third
## [1] "e" "f" "g"
```

```
x
```

```
## , , third = e
##
##      second
## first c d
##     a 1 3
##     b 2 4
```

- This is useful for subsetting, and for bookkeeping when you have data structured in a complicated multidimensional array (e.g. it is hard to remember what indexes the first vs second vs third dimensions without dimnames).
- A vector is not a matrix with 1 dimension, it has NULL dimsions.

```
x["a","c","g"] # easy subsetting
```

```
## [1] 9
```

```
z <- 1:3
dim(z)
```

```
## NULL
```

```
# Exercise: how do you get rid of the dimensions in following array?
x <- array(1:12, dim=c(2,2,3))
```

Vector attribute - `class`

- A special attribute named `class` is a key component of the "S3" object-oriented programming system (more on this later).
- Get and set with the `class()` function.
- You can set the class to `NULL` by `unclass()`.

```
lvec <- structure(list(age=54,hair="grey"),at1="abc",at2=1:3)
class(lvec) <- "prof"
lvec
```

```
## $age
## [1] 54
##
## $hair
## [1] "grey"
##
## attr(,"at1")
## [1] "abc"
## attr(,"at2")
## [1] 1 2 3
## attr(,"class")
## [1] "prof"
```

```
names(lvec)
```

```
## [1] "age"  "hair"
```

S3 atomic vectors

- The *class* of an object is an important attribute that control R's S3 system for object oriented programming.
- The class of an object will determine its behavior when you use that class in **generic** function such as `print()`, `summary()`.
  - A generic function is a function that has different behavior based on the class of the input.
- You could create your own S3 classes (chapter 13). Here, we will talk about some S3 classes that come with R by default.

- There are four important classes of atomic vector: factors, Dates, POSIXct and difftime. We will discuss factors. See the text, section 3.4 for information on the others.

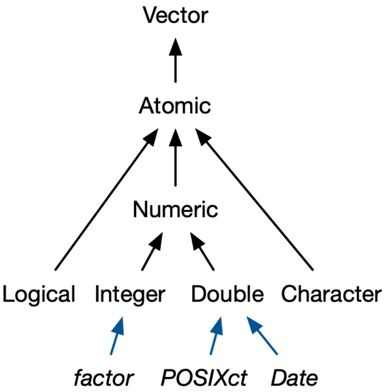


Figure 4: Vectors types in a tree

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
```

Subsetting vectors and extracting elements

- Subset with [:

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

```
## $names
## [1] "age"    "weight" "height" "hair"
```

```
attr(lvec,"at1") <- "abc" # add by name
attr(lvec,"at2") <- 1:3
attributes(lvec)
```

```
## $names
## [1] "age"    "weight" "height" "hair"
##
## $at1
## [1] "abc"
##
## $at2
## [1] 1 2 3
```

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

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

Take care using [[for lists

- [[ ]] (and $) can only return one item at a time. Trying to extract more than one has unexpected results.

```
lvec[[c(4,2)]] # the [[2]] element of lvec[[4]] !!
```

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

```
# Error when you do this: lvec$c("hair","weight")
```

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] <- 220
avec
```

```
## [1]  54 220  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"
```

```r
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
```

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

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

```r
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, data frames and tibbles

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
```

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

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] "double"
```

```
dim(A)
```

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

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

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

```r
str(A)
```

```
##  num [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
```

Row and column names

- Get and set with `rownames()` and `colnames()`, respectively.
- `names()` also gets and sets the column names

```
colnames(dd); names(dd)
```

```
## [1] "x" "y"
```

```
## [1] "x" "y"
```

```
rownames(dd) <- paste0("subj",1:nrow(dd))
rownames(dd)
```

```
## [1] "subj1" "subj2" "subj3" "subj4"
```

Subsetting data frames like a list

```r
dd$x
```

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

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

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

Subsetting and combining data frames like a matrix

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

```
##       x         y
## subj1 1 0.3735462
## subj2 2 2.1836433
```

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

```
##       x         y          z
## subj1 1 0.3735462 0.62911404
## subj2 2 2.1836433 0.06178627
## subj3 3 2.1643714 0.20597457
## subj4 4 5.5952808 0.17655675
```

tibbles

- A tidyverse replacement (or extension) of the data frame with different default behaviour.
- See section 3.6 for a discussion of some of the differences.

```
library(tibble)
tt <- tibble(x = 1:3, y = letters[1:3])
typeof(tt)
```

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

```
attributes(tt)
```

```
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
##
## $row.names
## [1] 1 2 3
##
## $names
## [1] "x" "y"
```

More on subsetting

- one-dimensional subsets of data frames are coerced to atomic vectors, but not so with tibbles.

```
dd[,1]; dd[1,1]
```

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

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

```
tt[,1]; tt[1,1]
```

```
## # A tibble: 3 x 1
##       x
##   <int>
## 1     1
## 2     2
## 3     3
```

```
## # A tibble: 1 x 1
##       x
##   <int>
## 1     1
```

List columns

- One way that tibbles improve on data frames is easier handling of columns whose elements are lists.

```
tibble(
  x = 1:3,
  y = list(1:2, 1:3, 1:4) # lists of different length on each subject
)
```

```
## # A tibble: 3 x 2
##       x y
##   <int> <list>
## 1     1 <int [2]>
## 2     2 <int [3]>
## 3     3 <int [4]>
```

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

## Warning in x && y: 'length(x) = 3 > 1' in coercion to 'logica

## Warning in x && y: 'length(x) = 3 > 1' in coercion to 'logica

## [1] FALSE

## [1] TRUE TRUE TRUE

## Warning in x || y: 'length(x) = 3 > 1' in coercion to 'logica
```

- 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
```