Data Modes and Structures in R

Data modes

Like any programming language, R, has multiple *data modes*. Unlike many other languages, R does not require designation of a variable's data mode prior to assignment. This is convenient, but also can cause problems when R assigns an unintended data mode to a variable. The common data modes in R include:

- numeric – real numbers (double) or integers
- complex – imaginary number
- logical – TRUE or FALSE; R also treats these as 1 and 0, respectively
- character – strings or text values

Data structures

Data of any of the modes described above is stored in one of R's *data structures*. We will primarily use four data structures this semester: vectors, matrices/arrays, lists, and data frames.

I. ***Vectors***: A vector is an R object that contains an ordered set of values. A vector is analogous to a row of data in Excel. Even variables assigned a single value are vectors of length 1. All values in a vector must be of the same data mode.

a. Creating vectors

-concatenate function: *c()*, combines specified values in a vector

```
> v=c(1,2,3,20)
```

-colon operator: *:*, generates an ordered sequence incremented by 1

```
> v=1:4
```

-sequence function: *seq()*, generates an ordered sequence incremented by the specified value or a sequence of the specified length

```
> v=seq(from=5,to=6,by=0.1)
> v=seq(from=-10,to=-5,length.out=10)
```

-repeat function: *rep()*, generates a vector of specified length containing the same value in each element of the vector

```
> v=rep(x=4,by=5)
```

-numeric function: *numeric()*, generates a vector of specified length filled with 0's

```
> v=numeric(length=5)
```

-vector function: *vector()*, generates a vector of specified length filled with FALSE's

```
> v=vector(length=5)
```

b. Useful vector functions

-mathematical operators: *+, -, *, /*

-logical operators: *<, >, <=, >=, ==, !=*

-*length()*: returns the length of the vector

-*max()*: returns the maximum value contained in the vector

-*min()*: returns the minimum value contained in the vector

-*sum()*: returns the sum of the values in the vector

-*cumsum()*: returns the cumulative sum for each element of the vector

-*mean()*: returns the mean of the vector

-*range()*: returns the minimum and maximum values

-*var()*: returns the variance of the vector

-*sd()*: returns the standard deviation of the vector

-*sort()*: returns a sorted version of the vector

-*order()*: returns the numerical indices of vector elements in sorted order

c. Vector indexing and subsetting

Because vectors are an ordered list, a single element or subset of elements can be referred to using square brackets, *[ ]*, and a numerical index.

```
> v=c(1,3,5,9,13)
> v[1]
[1] 1
> v[4]
[1] 9
> v[c(1,3,5)]
[1] 1 5 13
> v[-3]
[1] 1 3 9 13
```

An alternative means of indexing is a vector of logical values.

```
> v=c(1,3,5,9,13)
> v>3
[1] FALSE FALSE TRUE TRUE TRUE
>v[v>3]
[1] 5 9 13
```

The which() function creates numerical indices from a logical vector.

```
> v=c(1,3,5,9,13)
> v>3
[1] FALSE FALSE TRUE TRUE TRUE
> which(v>3)
[1] 3 4 5
```

The %in% function allows for logical indexing of overlapping sets.

```
> v=c(1,3,5,9,13)
> w=seq(from=1,to=21,by=4)
> v%in%w
[1] TRUE FALSE TRUE TRUE TRUE
> v[v!%in%w]
[1] 3
```

Note that you cannot delete an element from a vector, but you can reassign a subset of a vector to the same variable.

```
> v=c(1,3,5,9,13)
> v
[1] 1 3 5 9 13
> v=v[v>3]
> v
[1] 5 9 13
```

II. ***Matrices/Arrays***: A matrix is a vector with two additional attributes, the number of rows and columns. Matrices are a special case (two-dimensional) of an array. Matrices are analogous to Excel worksheets and a three-dimensional array is analogous to an Excel workbook. Because R views arrays as vectors with some extra description about their shape, nearly all functions that work on a vector will work on a matrix/array. However, many functions will only work with matrices. All values in a matrix must be the same data mode.

a. Creating matrices and arrays

-matrix function: *matrix()*, creates a matrix from a vector of values and dimensions

```
> A=matrix(0,nrow=2,ncol=2)
> A
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

-array function: *array()*, creates an array from a vector and dimensions

```
> B=array(rep(1:4,2),dim=c(2,2,2)
> B
, , 1
     [,1] [,2]
[1,]    1    2
[2,]    3    4
, , 2
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

b. Useful matrix functions

***many of these will work on higher dimensional arrays too

-*dim()*: returns the dimensions (number of rows and columns) of the matrix
-*nrow()*: returns the number of rows in the matrix
-*ncol()*: returns the number of columns in the matrix
-*rownames()*: returns the row names of the matrix; can also be used for assignment
-*colnames()*: returns the column names of the matrix; can also be used for assignment
-*rbind()*: add a vector to a specified matrix as a new row at the bottom of the matrix
-*cbind()*: add a vector to a specified matrix as a new column at the furthest right
-*%*%*: matrix multiplication
-*t()*: transpose the matrix
-*colMeans()*: calculate the mean of each column of the matrix
-*colSums()*: calculate the sum of each column of the matrix
-*apply()*: applies a function that works on a vector to each row or column of a matrix

c. Matrix and array indexing/subsetting

Just like vectors, elements or subsets of matrices and arrays can be indexed using square brackets. Because an array is a vector a single number can be used to index a particular element, but this would require a lot of mental math to get the desired element. Instead, we can use a numerical index along each dimension of an array, separated by a comma, to index a particular element or subset of elements.

```
> M=matrix(1:4,nrow=2,ncol=2)
> M[2,1]
[1] 2
> M[2,2]
[1] 4
> M[4]
[1] 4
> M[,2]
[1] 3 4
```

Logical values work for indexing with matrices too.

```
> M=cbind(1:5,c(3,9,15,25,76))
> M
      [,1] [,2]
[1,]    1    3
[2,]    2    9
[3,]    3   15
[4,]    4   25
[5,]    5   76
> M<8
     [,1]  [,2]
[1,] TRUE  TRUE
[2,] TRUE FALSE
[3,] TRUE FALSE
[4,] TRUE FALSE
[5,] TRUE FALSE
>M[M<8]
[1] 1 2 3 4 5 3
```

You can also subset a matrix based upon the content of a particular row or column.

```
> M=cbind(1:5,c(3,9,15,25,76))
> M[M[,1]<3,]
      [,1] [,2]
[1,]    1    3
[2,]    2    9
> M[M[,2]>24,1]
[1] 4 5
```

III. ***Lists***: A list in R is somewhat like a vector, but can accommodate values of different data mode, including other data structures. One could create a list of matrices or even a list of lists. It also is often indexed by element names rather than a numerical index. R's list data structure is very similar to a dictionary in Python or a Perl hash.

a. Creating lists

Lists are created using the *list()* function.

```
> a=list(name="walleye",length=225,tagged=TRUE)
> a
$name
[1] "walleye"

$length
[1] 225

$tagged
[1] TRUE
```

b. Useful list functions

-*length()*: returns the number of components in the list
-*names()*: returns the tags of a list
-*lapply()*: analogous to *apply()*, but operates on lists and returns a list
-*sapply()*: the same functionality as *lapply()*, but returns a matrix or vector

c. Indexing lists

There are a number of equivalent ways to access elements of a list, including element names or "tags" and square bracket (in this case double square brackets) indexing. For this reason, tags are optional, but tags do make referencing list components a lot easier.

```
> a=list(name="walleye",length=225,tagged=TRUE)
>  a$length
[1] 225
> a[["length"]]
[1] 225
> a[[2]]
[1] 225
```

d. Adding and deleting list elements

Additional elements can easily be added to an existing list.

```
> a=list(name="walleye",length=225)
> a
$name
[1] "walleye"

$length
[1] 225
> a$tagged=TRUE
> a
$name
[1] "walleye"

$length
[1] 225

$tagged
[1] TRUE
```

Elements can be removed from a list be setting them equal to NULL.

```
> a=list(name="walleye",length=225,tagged=TRUE)
> a$tagged=NULL
> a
$name
[1] "walleye"

$length
[1] 225
```

IV. ***Data frames***: A data frame is two-dimensional like a matrix, but can hold elements of different data modes. As matrices behave as vectors with a descriptor of shape, data frames behave as lists, but have two dimensions, which can often be very useful.

a. Creating data frames

Data frames are created by the *data.frame()* function. If we want to maintain character data as characters we must use the argument "stringsAsFactors=FALSE" in the *data.frame()* function. We'll discuss factors later this semester.

```
> names=c("walleye","perch")
> weight=c(270,57)
> d=data.frame(names,weight,stringsAsFactors=FALSE)
> d
       names weight
1    walleye    270
2      perch     57
```

b. Useful data frame functions

-*rbind()*: add a vector to a specified matrix as a new row at the bottom of the matrix
-*cbind()*: add a vector to a specified matrix as a new column at the furthest right
-*colMeans()*: calculate the mean of each column of the matrix
-*colSums()*: calculate the sum of each column of the matrix
-*apply()*: applies a function that works on a vector to each row or column of a matrix
-*merge()*: joins two data frames together using a shared column as an index
-*lapply()*: analogous to *apply()*, but operates on lists and returns a list
-*sapply()*: the same functionality as *lapply()*, but returns a matrix or vector

c. Indexing data frames

Because data frames behave like matrices and lists we can index subsets and elements of data frames in almost any fashion.

*** Code continued from "a" above

```
> d[[2]]
[1] 270 57
> d[,2]]
[1] 270 57
> d$names
[1] "walleye" "perch"
```