Part 2: Data Types and Manipulation

140.776 Statistical Computing

Ingo Ruczinski

Thanks to Thomas Lumley and Robert Gentleman of the R-core group (http://www.r-project.org/) for providing some tex files that appear in part in the following slides.

Types of data in R

- The basic data object is a vector of elements of type:

  numeric : numbers, either floating point or integer.
  character : each element is a character string.
  logical : each element is `TRUE` or `FALSE`.
  list : elements can be any type of object, including other lists.
- Components of the S language, such as functions, are also vectors.
- Any vector can include the missing data marker `NA` as an element.
- All vectors have a `length` and a `mode`. The functions `length` and `mode` return this information as does the `str` function.
- A structure consists of a data object plus additional information. Arrays and time series are examples of structures.

Variables

```
> x <- 5
> mode(x)
[1] "numeric"

> x <- "I like chocolate ice cream"
> sub("chocolate","strawberry",x)
[1] "I like strawberry ice cream"
> mode(x)
[1] "character"

> x <- LETTERS[1:5]
> x
[1] "A" "B" "C" "D" "E"
> mode(x)
[1] "character"

> x <- (1+2==4)
> x
[1] FALSE
> mode(x)
[1] "logical"
```

Generating simple vectors

- The assignment operator in R is the two-character sequence '`<-`'. An alternative is available, but its use is discouraged.
- Any type of vector can be created explicitly with the c (concatenation) function.
- Numeric vectors can be generated with the `seq` function or the sequence operator '`:`'.
- The function `rep` generates a new vector by repeating a vector of any mode (including a list) a specified number of times.
- Pseudo-random samples from various distributions can be created. The function names have the pattern `r<distname>`, such as `runif` for a uniform distribution or `rnorm` for a normal distribution.

Numeric vectors

```
> rn <- rnorm(100)
> str(rn)
 num [1:100] -0.696 -0.158 -2.449 -0.383  0.665 ...
> stem(rn)

  The decimal point is at the |

  -2 | 4
  -1 | 98754322111
  -0 | 8887776644333332222211111100
   0 | 000011112222333333334445555666777777788
   1 | 000001111223567899
   2 | 046

> rn <- rpois(100,lambda=3)
> table(rn)
rn
 0  1  2  3  4  5  6  7  8
 3 16 22 20 20 11  4  3  1
```

Characters

```
> rep(c("A","B"),4)
[1] "A" "B" "A" "B" "A" "B" "A" "B"

> rep(c("A","B"),rep(4,2))
[1] "A" "A" "A" "A" "B" "B" "B" "B"

> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
[15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

> LETTERS[c(1,20,7,3)]
[1] "A" "T" "G" "C"

> x <- c("A","T","G","C")
> x
[1] "A" "T" "G" "C"
```

Characters

```
> expand.grid(x,x,x)
   Var1 Var2 Var3
1     A    A    A
2     T    A    A
3     G    A    A
4     C    A    A
5     A    T    A
6     T    T    A
7     G    T    A
8     C    T    A
9     A    G    A
10    T    G    A
...
57    A    G    C
58    T    G    C
59    G    G    C
60    C    G    C
61    A    C    C
62    T    C    C
63    G    C    C
64    C    C    C
```

Character and logical vectors

```
> fabfive <- c("Karl","Rafa","Roger","Ingo","Brian")
> str(fabfive)
 chr [1:5] "Karl" "Rafa" "Roger" "Ingo" "Brian"

> fabfive < "K"
[1] FALSE FALSE FALSE  TRUE  TRUE

> grep("a",fabfive)
[1] 1 2 5

> grep("a",fabfive,value=T)
[1] "Karl"  "Rafa"  "Brian"

> grep("[a-e]",fabfive,value=T)
[1] "Karl"  "Rafa"  "Roger" "Brian"

> gsub("[a-e]","X",fabfive)
[1] "KXrl"  "RXfX"  "RogXr" "Ingo"  "XriXn"
```

Character and logical vectors

A list is an ordered collection of data of arbitrary types.

```
> krrib=list(name=c("Karl","Rafa","Roger","Ingo","Brian"),
+            age=c(17,20,18,19,5),dad=c(F,T,F,F,F))

> krrib
$name
[1] "Karl"  "Rafa"  "Roger" "Ingo"  "Brian"

$age
[1] 17 20 18 19  5

$dad
[1] FALSE  TRUE FALSE FALSE FALSE

> krrib$name
[1] "Karl"  "Rafa"  "Roger" "Ingo"  "Brian"
```

Disclaimer: ages are rough estimates only . . .

Factors

Qualitative data that can assume only a discrete set of values are represented by a `factor`.

```
> trt <- factor(rep(c("Control","Treated"),c(3,4)))
> str(trt)
 Factor w/ 2 levels "Control","Treated": 1 1 1 2 2 2 2

> summary(trt)
Control Treated
      3       4
```

If the levels of a factor are numeric (e.g. the treatments are labelled “1”, “2”, and “3”) it is important to ensure that the data are actually stored as a factor and not as numeric data. Always check this by using `summary`.

Factors

If you have numeric data that should be a factor, use `factor` or `as.factor` to convert it to a factor.

```
> x <- c(0,1,1,02,1,0,2,1,2)
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.000   1.000   1.000   1.111   2.000   2.000

> x <- as.factor(x)
> summary(x)
0 1 2
2 4 3
```

Ordered Factors

An ordered factor is, not surprisingly, a special type of factor in which the levels have an ordering.

```
> pain <- ordered(c("Moderate","None","Severe","Severe","None"),
+                 levels = c("None", "Moderate", "Severe"))
> str(pain)
 Ord.factor w/ 3 levels "None"<"Moderate"<..: 2 1 3 3 1

> pain
[1] Moderate None     Severe   Severe   None
Levels: None < Moderate < Severe

> summary(pain)
    None Moderate   Severe
       2        1        2
```

Ordered Factors

```
> class(pain)
[1] "ordered" "factor"

> mode(pain)
[1] "numeric"

> typeof(pain)
[1] "integer"
```

You do not want the following:

```
> pain <- ordered(c("Moderate", "None", "Severe", "Severe", "None"))
> pain
[1] Moderate None     Severe   Severe   None
Levels: Moderate < None < Severe
```

Data frames

A `data.frame` is the basic S structure for a data set that can be represented as a set of observations (rows) on several variables (columns). Most of the data sets you see listed in the output of `data()` are data frames.

```
> data(Formaldehyde)
> str(Formaldehyde)
'data.frame':   6 obs. of  2 variables:
 $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
> summary(Formaldehyde)
      carb           optden
 Min.   :0.1000   Min.   :0.0860
 1st Qu.:0.3500   1st Qu.:0.3132
 Median :0.5500   Median :0.4920
 Mean   :0.5167   Mean   :0.4578
 3rd Qu.:0.6750   3rd Qu.:0.6040
 Max.   :0.9000   Max.   :0.7820
```

Data frames

Columns in data frames are usually numeric variables or factors. (Other possibilities exist but are rare.) Always check a data frame using `summary` to ensure that variables that should be factors are factors. Factors are summarized by frequency tables.

```
> data(iris); summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length
 Min.   :4.300   Min.   :2.000   Min.   :1.000
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
 Median :5.800   Median :3.000   Median :4.350
 Mean   :5.843   Mean   :3.057   Mean   :3.758
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
 Max.   :7.900   Max.   :4.400   Max.   :6.900
  Petal.Width          Species
 Min.   :0.100   setosa    :50
 1st Qu.:0.300   versicolor:50
 Median :1.300   virginica :50
 Mean   :1.199
 3rd Qu.:1.800
 Max.   :2.500
```

Extracting subsets

One of the keys to mastering the S language is learning to use the extraction (or subset) operators effectively.

A typical message on the R-help email list asks:

Is there a better solution to select rows from a data.frame than by iterating through the whole set and evaluating every case one by one?

Is there maybe something like:

```
d<-data.frame(...)
maleOver40<-select.data.frame(d,"( sex=m or sex=M) and age > 40)")
```

Of course, I could use direct database-requests, but this would require my data to be stored in a database...

The general subset operator

The '[' operator is the general extraction operator. It creates an object of the same mode as the object to which it is applied. In the expression `x[i]` several forms of indices `i` can be used:

positive integers:
indicate the positions of the elements to extract. The first position is numbered 1.

negative integers:
indicate all elements except those at indices numbered `-i`. That is, `x[-1]` means "drop the first element of `x`".

logical vectors:
if `i` is a logical vector of the same length as `x` then the elements of `x` corresponding to `TRUE` in `i` are returned.

character variables:
are matched against the names of elements of `x`.

Examples of the general subset

```
> rn <- rnorm(100)
> rn[1:3]
[1] -1.52057659 -0.29059035 -0.08113082
> rn[3:1]
[1] -0.08113082 -0.29059035 -1.52057659

> rn[98:100]
[1] 0.2454289 0.1317154 1.1490626
> rn[-(1:97)]
[1] 0.2454289 0.1317154 1.1490626

> str(rn[rn>0])
 num [1:53] 0.254 0.776 1.323 0.239 0.510 ...

> con <- c(e=exp(1),pi=pi,twopi=2*pi)
> con[c("e","pi")]
       e       pi
2.718282 3.141593
```

Extracting single elements

The '[' operator returns an object of the same mode as the object to which it is applied. The '[[' and '$' operators extract single elements in their native mode. The distinction is like that between "an element of a set" (what '[[' produces) and "a subset of size 1 from a set", (what '[' produces).

```
> li <- list(pi=pi,e=exp(1))
> mode(li[1])
[1] "list"
> mode(li[[1]])
[1] "numeric"
> li[1]
$pi
[1] 3.141593

> sqrt(li[1])
Error in sqrt(li[1]) : Non-numeric argument to mathematical function
> sqrt(li[[1]])
[1] 1.772454
```

Subsets applied to data frames

Data frames are most naturally regarded as a rectangular structure. We can use '[' to extract subsets of rows or subsets of columns or both. For this, two indexing expressions are used. Omitting an indexing expression for the rows (or columns) means to use all the rows (or columns).

```
> dim(iris)
[1] 150   5
> summary(iris[,c(1,2,5)])
  Sepal.Length    Sepal.Width          Species
 Min.   :4.300   Min.   :2.000   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   versicolor:50
 Median :5.800   Median :3.000   virginica :50
 Mean   :5.843   Mean   :3.057
 3rd Qu.:6.400   3rd Qu.:3.300
 Max.   :7.900   Max.   :4.400
> dim(iris[iris$Species=="setosa",])
[1] 50  5
```

‘subset’ and the %in% operator

Extracting rows from a data frame based on values of some of the variables is common. But for example, writing expressions like `iris[iris$Species == "setosa", ]` can become tedious. The `subset` function is an alternative.

```
> dim(subset(iris,Species=="setosa"))
[1] 50  5
```

Sometimes we want to select for several possible values of a variable. A general function `match` returns the indices generated by matching one set of values against another another. It is often used to find out which values are present in another set. The operator `%in%` phrases this more succinctly.

```
> dim(subset(iris,Species%in%c("setosa","versicolor")))
[1] 100   5
```

The selection question from R-help

Recall the question asked on the R-help email list:

Is there maybe something like:

```
d<-data.frame(...)
maleOver40<-select.data.frame(d,"( sex=m or sex=M) and age > 40)")
```

Peter Dalgaard answered:

```
> maleOver40 <- subset(d, sex \%in\% c(’m’, ’M’) & age > 40)
```

match

```
match                   package:base                   R Documentation

Value Matching

Description:
     'match' returns a vector of the positions of (first) matches of
     its first argument in its second.
     '%in%' is a more intuitive interface as a binary operator, which
     returns a logical vector indicating if there is a match or not for
     its left operand.

Usage:
     match(x, table, nomatch = NA, incomparables = FALSE)
     x %in% table

Arguments:
       x: the values to be matched.
   table: the values to be matched against.
 nomatch: the value to be returned in the case when no match is found.
          Note that it is coerced to 'integer'.
```

Subsets that are larger than the original

The extraction operator '[' is more general than a subset operator. By repeating indices we can produce "subsets" that are larger than the original.

```
> c("Yes","No")
[1] "Yes" "No"

> rep(c("Yes","No"),3)
[1] "Yes" "No"  "Yes" "No"  "Yes" "No"

> rep(1:2,3)
[1] 1 2 1 2 1 2

> c("Yes","No")[rep(1:2,3)]
[1] "Yes" "No"  "Yes" "No"  "Yes" "No"

> LETTERS[c(11,18,18,9,2)]
[1] "K" "R" "R" "I" "B"
```

More on single element extraction

The '[[' and '$' operators both extract a single element in its native mode. The '[[' operator requires an index position or a (quoted) name. The '$' operator requires a name without quotes. The expression x$nm is simply a short form for x[["nm"]].

```
> data(Formaldehyde); str(Formaldehyde)
'data.frame':   6 obs. of  2 variables:
 $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782

> Formaldehyde$optden
[1] 0.086 0.269 0.446 0.538 0.626 0.782

> Formaldehyde[["optden"]]
[1] 0.086 0.269 0.446 0.538 0.626 0.782

> Formaldehyde[[2]]
[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

Single element extraction & data frames

Notice that we can use only one index without extra commas to access a column in a data frame.

```
> Formaldehyde[[2]]
[1] 0.086 0.269 0.446 0.538 0.626 0.782

> Formaldehyde[,2]
[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

Single element extraction returns a column because a data frame is in fact a list of columns, as indicated by the result of the command `str(Formaldehyde)`. The `[row,col]` form of indexing shown previously is a special indexing method that makes a data frame appear to be rectangular.

Dropping dimensions

Another peculiarity in the previous example is having the single column accessed as `frm[,j]` returned as a vector, not as a data frame. Recall that '`[`' is supposed to return an object of the same mode as the original object.

There is an exception: single dimensions in subarrays are dropped. Adding `drop = FALSE` to an extraction expression suppresses this.

```
> str(Formaldehyde[,2])
 num [1:6] 0.086 0.269 0.446 0.538 0.626 0.782

> str(Formaldehyde[,2,drop=FALSE])
'data.frame':   6 obs. of  1 variable:
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
```

NA - the missing data marker

The codes `NA` (not available) and `NaN` (not a number) indicates a missing data value in a vector or other data structure. Both are called NA's. An NA can be part of the original data, or it can be the result of operations on other data where the result is undefined, or it can be assigned.

```
> rrn <- rnorm(100)
> lrn <- log(rrn)
Warning message:
NaNs produced in: log(x)
> str(rrn)
 num [1:100]  0.198  0.261  1.647  1.679 -2.463 ...
> str(lrn)
 num [1:100] -1.617 -1.344  0.499  0.518    NaN ...

> NA & TRUE
[1] NA
> NA | TRUE
[1] TRUE
```

Use is.na to check for missing data

Note that we check for missing data with `is.na`. This is the only way to detect missing data. A common mistake is trying to check for missing data with expressions like `x == NA`. This doesn't work as expected. Missing values propagate in operations, including comparison operations. Comparing another value to `NA` always produces an `NA`.

```
> str(1 + lrn)
 num [1:100] -0.617 -0.344  1.499  1.518    NaN ...

> lrn.msng <- lrn == NA
> str(lrn.msng)
 logi [1:100] NA NA NA NA NA NA NA NA NA NA NA NA ...

> lrn.msng <- is.na(lrn)
> str(lrn.msng)
 logi [1:100] FALSE FALSE FALSE FALSE  TRUE  TRUE ...
```

Summaries of data that have NA's

Applying a summary function, such as `mean`, `median`, or `var` to data with any `NA`'s (or `NaN`'s) will return `NA` (or `NaN`).

If you want the value of the summary function after excluding the NA's, you must exclude the NA's then do the summary. Several summary functions allow an argument `na.rm = TRUE` that causes this to be done automatically.

```
> mean(lrn)
[1] NaN

> mean(lrn[!is.na(lrn)])
[1] -0.5095632

> mean(lrn,na.rm=TRUE)
[1] -0.5095632
```

Other special numeric values

NA's are allowed in all types of data. Numeric data also allows `NaN`, as shown previously, `Inf` (∞) and `-Inf` ($-\infty$).

```
> log(0:2)
[1]      -Inf 0.0000000 0.6931472
> exp(log(0:2))
[1] 0 1 2
```

There are ways of detecting `NaN` and infinite values.

```
> x=rnorm(5)
> x
[1]  1.0847354 -0.2244801 -0.3103911 -0.6022185  0.5310318

> y=log(x)
Warning message:
NaNs produced in: log(x)

> is.nan(y)
[1] FALSE  TRUE  TRUE  TRUE FALSE
```

Complex numbers

A related issue is arithmetic on complex values. By default, arithmetic on numeric data is done as real (i.e. floating point) numbers. Complex values can be created by designating an imaginary part, in which case complex arithmetic is used.

```
> str(sqrt(-2:2))
 num [1:5]  NaN  NaN 0.00 1.00 1.41
Warning message:
NaNs produced in: sqrt(-2:2)

> (-2:2) + 0i
[1] -2+0i -1+0i  0+0i  1+0i  2+0i

> str(sqrt((-2:2) + 0i))
 cplx [1:5] 0+1.41i 0+1.00i 0+0.00i ...
```

The recycling rule

If a and b are vectors of the same length n then a*b is the element-by-element product. But what if they are not the same length?

- It is clear that 2*b should be the vector whose elements are twice those of b, so the 2 must be repeated n times.
- Generalising this, the shorter of the two arguments is always repeated to make it as long as the longer argument. If this is not an exact multiple a warning is given.

```
> 1+1:6
[1] 2 3 4 5 6 7
> 1:2+1:6
[1] 2 4 4 6 6 8
> 1:4+1:6
[1] 2 4 6 8 6 8
Warning message:
longer object length
        is not a multiple of shorter object length in: 1:4 + 1:6
```

Matrices and other arrays

R provides several linear algebra operations on matrices. Although matrices and data frames seem similar, their underlying structure is different. Matrices are homogeneous (i.e. all elements are the same type) whereas data frames can be heterogeneous with elements of different types - numeric, factors, ordered factors, . . .

Matrices are created with the `matrix` or `array` functions. They are stored in *column major* ordering, which means the first column, followed by the second column, etc. The `array` function can be used to create multi-dimensional arrays.

```
> str(rmat<-matrix(rnorm(8),nrow=2))
 num [1:2, 1:4] -0.842  0.642 -0.184 -0.386 -2.108 ...

> rmat
           [,1]       [,2]       [,3]       [,4]
[1,] -0.8419032 -0.1836398 -2.1084794 -0.1035748
[2,]  0.6415143 -0.3863284 -0.3363234 -1.6427572
```

Grouped data

Assume the file `fermentation.txt` contains data on the effect of oxygen levels on the fermentation end product.

```
ethanol oxygen sugar
0.59 0 Galactose
0.30 0 Galactose
0.25 0 Glucose
0.03 0 Glucose
0.44 46 Galactose
0.18 46 Galactose
...
```

You can create a `grouped data` object:

```
fermentation <- groupedData(ethanol~oxygen|sugar,
                    data=read.table("fermentation.txt",header=T),
                    labels=list(x="Oxygen",y="Ethanol"))
```

Grouped data

```
> library("nlme")
> trellis.device(col=F)
> plot(fermentation)
```