Introduction to R

Educational Materials
©2007 S. Falcon, R. Ihaka, and R. Gentleman
Data Structures

- R has a rich set of *self-describing* data structures.

```r
> class(z)
[1] "character"

> class(x)
[1] "data.frame"

> x[1:2, ]

    type  time
 1 case 0.6291721
 2 case 0.1190050
```

- There is no need to declare the types of the variables.
Data Structures (continued)

- **vector** - arrays of the same type
- **list** - can contain objects of different types
- **environment** - hashtable
- **data.frame** - table-like
- **factor** - categorical
- **Classes** - arbitrary record type
- **function**
Atomic Data Structures

• In R, the basic data types are vectors, not scalars.

• A vector contains an indexed set of values that are all of the same type:
  – logical
  – numeric
  – complex
  – character

• The numeric type can be further broken down into integer, single, and double types (but this is only important when making calls to foreign functions, eg. C or Fortran.)
Creating Vectors

There are two symbols that can be used for assignment: <- and =.

> v <- 123
[1] 123

> s <- "a string"
[1] "a string"

> t <- TRUE
[1] TRUE

> length(letters)
[1] 26

> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
[17] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
Functions for Creating Vectors

- **c** - concatenate
- **:** - integer sequence, **seq** - general sequence
- **rep** - repetitive patterns
- **vector** - vector of given length with default value

```r
> seq(1, 3)
[1] 1 2 3
```
```r
> 1:3
[1] 1 2 3
```
```r
> rep(1:2, 3)
[1] 1 2 1 2 1 2
```
```r
> vector(mode="character", length=5)
[1] "" "" "" "" ""
```
Matrices and $n$-Dimensional Arrays

- Can be created using matrix and array.
- Are represented as a vector with a dimension attribute.
- Left most index is fastest (like Fortran or Matlab)
Matrix Examples

> x <- matrix(1:10, nrow=2)
> dim(x)

[1] 2 5

> x

[1,] 1 3 5 7 9
[2,] 2 4 6 8 10

> as.vector(x)

[1] 1 2 3 4 5 6 7 8 9 10
Naming

The elements of a vector can (and often should) be given names. Names can be specified

- at creation time
- later by using `names`, `dimnames`, `rownames`, `colnames`

```r
> x <- c(a=0, b=2)
> x
a b
0 2

> names(x) <- c("Australia", "Brazil")
> x
Australia Brazil
 0 2
```
Naming (continued)

> x <- matrix(c(4, 8, 5, 6, 4, 2, 1, 5, 7), nrow=3)
> dimnames(x) <- list(
+  year = c("2005", "2006", "2007"),
+  "mode of transport" = c("plane", "bus", "boat"))
> x

    mode of transport
year  plane bus boat
2005   4   6   1
2006   8   4   5
2007   5   2   7
Subsetting

- One of the most powerful features of R is its ability to manipulate subsets of vectors and arrays.

- Subsetting is indicated by $[ ]$.

- Note that $[ ]$ is actually a function (try `get("["")`). $x[2, 3]$ is equivalent to "["(x, 2, 3). Its behavior can be customized for particular classes of objects.

- The number of indices supplied to $[ ]$ must be either the dimension of $x$ or 1.
Subsetting with Positive Indices

- A subscript consisting of a vector of positive integer values is taken to indicate a set of indices to be extracted.

```r
> x <- 1:10
> x[2]
[1] 2
> x[1:3]
[1] 1 2 3
```

- A subscript which is larger than the length of the vector being subsetted produces an NA in the returned value.

```r
> x[9:11]
[1] 9 10 NA
```
Subsetting with Positive Indices (continued)

- Subscripts which are zero are ignored and produce no corresponding values in the result.
  
  ```r
  x[0:1]
  ```
  ```r
  [1] 1
  ```
  ```r
  > x[c(0, 0, 0)]
  ```
  ```r
  integer(0)
  ```

- Subscripts which are NA produce an NA in the result.
  ```r
  > x[c(10, 2, NA)]
  ```
  ```r
  [1] 10 2 NA
  ```
Assignments with Positive Indices

- Subset expressions can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

  ```
  > x[2] <- 200
  > x[8:10] <- 10
  > x
  [1] 1 200 3 4 5 6 7 10 10 10
  ```

- If a zero or NA occurs as a subscript in this situation, it is ignored.
Subsetting with Negative Indexes

- A subscript consisting of a vector of negative integer values is taken to indicate the indices which are not to be extracted.

  \[ x[-(1:3)] \]

  > x[-(1:3)]

  [1] 4 5 6 7 10 10 10

- Subscripts which are zero are ignored and produce no corresponding values in the result.

- \texttt{NA} subscripts are not allowed.

- Positive and negative subscripts cannot be mixed.
Assignments with Negative Indexes

- Negative subscripts can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

\[
\begin{align*}
> & \ x = 1:10 \\
> & \ x[-(8:10)] = 10 \\
> & \ x \\
[1] & \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 10 \ 8 \ 9 \ 10
\end{align*}
\]

- Zero subscripts are ignored.

- NA subscripts are not permitted.
Subsetting by Logical Predicates

- Vector subsets can also be specified by a logical vector of `TRUE` and `FALSE`.

```r
> x = 1:10
> x > 5
[1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> x[x > 5]
[1] 6 7 8 9 10
```

- NA values used as logical subscripts produce NA values in the output.

- The subscript vector can be shorter than the vector being subsetted. The subscripts are recycled in this case.

- The subscript vector can be longer than the vector being subsetted. Values selected beyond the end of the vector produce NAs.
Subsetting by Name

• If a vector has named elements, it is possible to extract subsets by specifying the names of the desired elements.

\[
> x \leftarrow \text{c}(a=1,\ b=2,\ c=3) \\
> x[\text{c}("c",\ "a",\ "foo")]
\]

\[
\begin{array}{ccc}
  c & a & <\text{NA}> \\
  3 & 1 & \text{NA}
\end{array}
\]

• If several elements have the same name, only the first of them will be returned.

• Specifying a non-existent name produces an NA in the result.
Subsetting matrices

- when subsetting a matrix, missing subscripts are treated as if all elements are named; so $x[1,]$ corresponds to the first row and $x[,3]$ to the third column.

- for arrays, the treatment is similar, for example $y[,,1]$.

- these can also be used for assignment, $x[1,]=20$
Subsetting Arrays

- Rectangular subsets of arrays obey similar rules to those which apply to vectors.

- One point to note is that arrays can also be treated as vectors. This can be quite useful.

```r
> x = matrix(1:9, ncol=3)
> x[ x > 6 ]
[1] 7 8 9
> x[row(x) > col(x)] = 0
> x

[,1] [,2] [,3]
[1,] 1 4 7
[2,] 0 5 8
[3,] 0 0 9
```
Custom Subsetting Example

```r
> library("Biobase")
> data(sample.ExpressionSet)
> class(sample.ExpressionSet)

[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"

> dim(sample.ExpressionSet)

Features  Samples
  500       26

> slotNames(sample.ExpressionSet)

[1] "assayData"       "phenoData"        "featureData"
[4] "experimentData" "annotation"        ".__classVersion__"
```
Custom Subsetting Example

> sample.ExpressionSet

ExpressionSet (storageMode: lockedEnvironment)
assayData: 500 features, 26 samples
  element names: exprs, se.exprs
phenoData
  sampleNames: A, B, ..., Z (26 total)
  varLabels and varMetadata description:
    sex: Female/Male
    type: Case/Control
    score: Testing Score
featureData
  featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at, ..., 31739_at
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
Annotation: hgu95av2
Custom Subsetting Example

\[
> \text{sample.ExpressionSet}[1:2, 2:5]
\]

ExpressionSet (storageMode: lockedEnvironment)
assayData: 2 features, 4 samples
  element names: exprs, se.exprs
phenoData
  sampleNames: B, C, D, E
  varLabels and varMetadata description:
    sex: Female/Male
    type: Case/Control
    score: Testing Score
featureData
  featureNames: AFFX-MurIL2_at, AFFX-MurIL10_at
  fvarLabels and fvarMetadata description: none
experimentData: use 'experimentData(object)'
Annotation: hgu95av2
Vectorized Arithmetic

- Most arithmetic operations in the R language are *vectorized*. That means that the operation is applied element-wise.
  
  ```r
  > 1:3 + 10:12
  [1] 11 13 15
  ```

- In cases where one operand is shorter than the other the short operand is recycled, until it is the same length as the longer operand.
  
  ```r
  > 1 + 1:5
  [1] 2 3 4 5 6
  ```

  ```r
  > paste(1:5, "A", sep="")
  [1] "1A" "2A" "3A" "4A" "5A"
  ```

- Many operations which need to have explicit loops in other languages do not need them with R. You should vectorize any functions you write.
Lists

• In addition to atomic vectors, R has a number of recursive data structures. Among the important members of this class are lists and environments.

• A list is an ordered set of elements that can be arbitrary R objects (vectors, other lists, functions, ...). In contrast to atomic vectors, which are homogeneous, lists and environments can be heterogeneous.

```r
> lst = list(a=1:3, b = "ciao", c = sqrt)
> lst
$a
[1] 1 2 3

$b
[1] "ciao"

$c
function (x) .Primitive("sqrt")
> lst$c(81)
[1] 9
```
Environments

- One difference between lists and environments is that there is no concept of ordering in an environment. All objects are stored and retrieved by `name`.

```r
> e1 = new.env()
> e1[["a"]]<-1:3
> assign("b", "ciao", e1)
> ls(e1)
[1] "a" "b"
```

- Random access to large environment can be sped up by using hashing (see the manual page of `new.env`).

- Names must match exactly (for lists, partial matching is used for the `$` operator).
Subsetting and Lists

- Lists are useful as containers for grouping related things together (many R functions return lists as their values).
- Because lists are a recursive structure it is useful to have two ways of extracting subsets.
- The [ ] form of subsetting produces a sub-list of the list being subsetted.
- The [[ ]] form of subsetting can be used to extract a single element from a list.
List Subsetting Examples

• Using the [ ] operator to extract a sublist.
  > lst[1]
  $a$
  [1] 1 2 3

• Using the [[ ]] operator to extract a list element.
  > lst[[1]]
  [1] 1 2 3

• As with vectors, indexing using logical expressions and names is also possible.
List Subsetting by Name

- The dollar operator provides a short-hand way of accessing list elements by name. This operator is different from all other operators in R, it does not evaluate its second operand (the string).

  ```r
  > lst$a
  [1] 1 2 3
  > lst[["a"]]
  [1] 1 2 3
  ```

- For $ partial matching is used, for [[] it is not by default, but can be turned on.
Accessing Elements in an Environment

- Access to elements in environments can be through, `get`, `assign`, `mget`.
- You can also use the dollar operator and the `[[ ]]` operator, with character arguments only. No partial matching is done.

```r
> e1$a
[1] 1 2 3
> e1[["b"]]
[1] "ciao"
```
Assigning values in Lists and Environments

- Items in lists and environments can be (re)placed in much the same way as items in vectors are replaced.

```r
> lst[[1]] = list(2,3)
> lst[[1]]

[[1]]
[1] 2

[[2]]
[1] 3

> e1$b = 1:10
> e1$b

[1] 1 2 3 4 5 6 7 8 9 10
```
Data Frames

- Data frames are a special R structure used to hold a set of spreadsheet like table. In a `data.frame`, the observations are the rows and the covariates are the columns.

- Data frames can be treated like matrices and be indexed with two subscripts. The first subscript refers to the observation, the second to the variable.

- Data frames are really lists, and list subsetting can also be used on them.
Data Frames (continued)

```r
> df <- data.frame(type=rep(c("case", "control"), c(2, 3)), time=rexp(5))
> df

          type time
1       case 1.610914
2       case 0.721062
3     control 1.577255
4     control 1.873261
5     control 2.059024

> df$time

[1] 1.610914 0.721062 1.577255 1.873261 2.059024

> names(df)

[1] "type" "time"

> rn <- paste("id", 1:5, sep="")
> rownames(df) <- rn
> df[1:2, ]

          type time
id1    case 1.610914
id2    case 0.721062
```
Getting Help

There are a number of ways of getting help:

- `help.start` and the HTML help button in the Windows GUI
- `help` and `?: help("data.frame")`
- `help.search`, `apropos`
- `RSiteSearch` (requires internet connection)
- Online manuals
- Mailing lists
Packages

- In R the primary mechanism for distributing software is via *packages*.
- CRAN is the major repository for packages.
- You can either download packages manually or use `install.packages` or `update.packages` to install and update packages.
- In addition, on Windows and other GUIs, there are menu items that facilitate package downloading and updating.
- It is important that you use the R package installation facilities. You cannot simply unpack the archive in some directory and expect it to work.
Packages - Bioconductor

- Bioconductor packages are hosted in CRAN-style repositories and are accessible using \texttt{install.packages}.

- The most reliable way to install Bioconductor packages (and their dependencies) is to use \texttt{biocLite}.

- Bioconductor has both a release branch and a development branch. Each Bioconductor release is compatible with its contemporary R release.

- Bioconductor packages have vignettes.
Name spaces

• Having many more packages, written by many different people, can cause some problems.

• When packages are loaded into R, they are essentially attached to the `search` list, see `search`.

• This creates the possibility of variable masking: the same name being for different functions in different packages.

• Name spaces were introduced in R 1.7.0 to alleviate the problem.
Control-Flow

R has a standard set of control flow functions:

- Looping: `for`, `while` and `repeat`.
- Conditional evaluation: `if` and `switch`.
Two Useful String Functions

1. Concatenate strings: \texttt{paste}

2. Search strings: \texttt{grep}
Example: paste

```r
> s <- c("apple", "banana", "lychee")
> paste(s, "X", sep="_")

[1] "apple_X"  "banana_X"  "lychee_X"

> paste(s, collapse="", ")

[1] "apple, banana, lychee"
```
Example: grep

```r
> library("ALL")
> data(ALL)
> class(ALL$mol.biol)

[1] "factor"

> negIdx <- grep("NEG", ALL$mol.biol)
> negIdx[1:10]

[1]  2  5  6  7  8  9 12 14 16 21
```
The apply Family

- A natural programming construct in R is to apply the same function to elements of a list, of a vector, rows of a matrix, or elements of an environment.

- The members of this family of functions are different with regard to the data structures they work on and how the answers are dealt with.

- Some examples, apply, sapply, lapply, mapply, eapply.
apply

• apply applies a function over the margins of an array.

• For example,
  
  > apply(x, 2, mean)
  
  computes the column means of a matrix x, while

  > apply(x, 1, median)
  
  computes the row medians.
apply

apply is usually not faster than a for loop. But it is more elegant.

```r
> a=matrix(runif(1e6), ncol=10)

> system.time({
+   s1 = apply(a, 1, sum)
+ })

     user     system    elapsed
   1.186     0.028     1.215

> system.time({
+   s2 = numeric(nrow(a))
+   for(i in 1:nrow(a))
+     s2[i] = sum(a[i,])
+ })

     user     system    elapsed
   0.667     0.007     0.673

See also: rowSums and colSums.
```
Writing Functions

- Writing R functions provides a means of adding new functionality to the language.
- Functions that a user writes have the same status as those which are provided with R.
- Reading the functions provided with the R system is a good way to learn how to write functions.
A Simple Function

• Here is a function that computes the square of its argument.
  
  ```
  > square = function(x) x*x
  > square(10)
  [1] 100
  ```

• Because the function body is vectorized, so is this new function.
  
  ```
  > square(1:4)
  [1]  1  4  9 16
  ```
Composition of Functions

• Once a function is defined, it is possible to call it from other functions.

  > sumsq = function(x) sum(square(x))
  > sumsq(1:10)

[1] 385
Returning Values

- Any single R object can be returned as the value of a function; including a function.

- If you want to return more than one object, you should put them in a list (usually with names), or an S4 object, and return that.

- The value returned by a function is either the value of the last statement executed, or the value of an explicit call to `return`.

- `return` takes a single argument, and can be called from any where in a function.
Control of Evaluation

- In some cases you want to evaluate a function that may fail, but you do not want to get stuck with an error.
- In these cases the function `try` can be used.
- `try(expr)` will either return the value of the expression `expr`, or an object of class `try-error`
- `tryCatch` provides a more configurable mechanism for condition handling and error recovery.
Object Oriented Programming

- Object oriented programming is a style of programming where one attempts to have software reflections ("models") of application-oriented concepts and to write functions (methods) that operate on these objects.

- The R language has two different object oriented paradigms, one S3 is older and should not be used for new projects. The second, S4 is newer and is currently under active development.

- These objects systems are more like OOP in Scheme, Lisp or Dylan than they are like OOP in Java or C++.
Classes

- In OOP there are two basic ingredients, objects and methods.
- An object is an instance of a class, and all objects of a particular class have some common characteristics.
- Inheritance or class extension: Class B is said to extend class A if a member of B has all the attributes that a member of A does, plus some other attributes.
Generic Functions

- A *generic function* is a dispatcher that examines the classes(!) of its arguments and invokes the most appropriate specific method.

- Methods are “normal” functions that are registered with generic functions, by indicating their existence together with the number and classes of its arguments (its “signature”).

- In the previous example, if a generic function is called with an instance of class B and there is no class B method, a class A method could be used.
Classes

- A class consists of a set of *slots* each containing a specific type (character, numeric, etc.).
- *Methods* can be defined for classes. A rectangle class that has slots for length and width could have an *area* method.
- Slots are accessed using @, but accessor methods are preferred.
Classes (S4 example)

> setClass("Person", representation(name="character",
+           height="numeric",
+           country="character"))

[1] "Person"

> p <- new("Person", name="Alice", height=5.0, country="UK")
> p

An object of class "Person"
Slot "name":
[1] "Alice"

Slot "height":
[1] 5

Slot "country":
[1] "UK"

> p@name

[1] "Alice"
S3

- S3 OOP no real mechanism for making sure that objects from a specific class have anything in common - it is just expected.

- One can make any object an instance of class *foo*, by assigning a class attribute, \( \text{class(x)} = \"foo\". \)

- S3 handles inheritance by setting several different class attributes (but this can lead to confusion).

- S3 is not suitable for complicated or multi-author projects.
References


