Introduction to R

Educational Materials

Modified 21 November, 2009, M. Morgan

Getting Help from Within R

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

Data Structures

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

- `vector` - arrays of the same type
- `list` - can contain objects of different types
- `environment` - hash table
- `data.frame` - table-like
- `factor` - categorical
- Classes - arbitrary record type
- `function`

Vectors

- 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*

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

> letters             # 'letters' is a built-in variable

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
[16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

> length(letters)     # 'length' is a function

[1] 26
```

Functions for Creating Vectors

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

```
> seq(1, 3)
[1] 1 2 3
> 1:3
[1] 1 2 3
> rep(1:2, 3)
[1] 1 2 1 2 1 2
> 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] [,2] [,3] [,4] [,5]
[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`

```
> 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.

```
> 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 subset produces an NA in the returned value.

```
>  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.

```
> x[0:1]
[1] 1
> x[c(0, 0, 0)]
integer(0)
```

- Subscripts which are `NA` produce an `NA` in the result.

```
> x[c(10, 2, NA)]
[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)]
[1]  4  5  6  7 10 10 10
```

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

```
> x = 1:10
> x[-(8:10)] = 10
> x
 [1] 10 10 10 10 10 10 10  8  9 10
```

- 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`s and `FALSE`s.

```
> x = 1:10
> x > 5
 [1] FALSE FALSE FALSE FALSE FALSE  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 `NA`s.

Subsetting by Name

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

```
> x <- c(a=1, b=2, c=3)
> x[c("c", "a", "foo")]
   c    a <NA>
   3    1   NA
```

- 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.

```
> 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
```

Vectorized Arithmetic

- Most arithmetic operations in the R language are *vectorized*. That means that the operation is applied element-wise.

```
> 1:3 + 10:12
[1] 11 13 15
```

- When one operand is shorter than the other, the short operand is recycled until it is the same length as the longer operand.

```
>  1 + 1:5
[1] 2 3 4 5 6
>  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.

```
> 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**.

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

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

Subsetting and Lists

- Lists are useful as containers for grouping related thing 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).

```
> 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`.

```
> mget(c("a", "b"), e1)
$a
[1] 1 2 3

$b
[1] "ciao"
```

- You can also use the dollar operator and the `[[ ]]` operator, with character arguments only. No partial matching is done.

```
> 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.

```
> 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)

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

     type      time
1    case 0.9921268
2    case 0.4827009
3 control 0.9276607
4 control 2.3399579
5 control 0.8607080

> df$time

[1] 0.9921268 0.4827009 0.9276607 2.3399579 0.8607080

> names(df)

[1] "type" "time"

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

    type      time
id1 case 0.9921268
id2 case 0.4827009
```

Packages

- In R the primary mechanism for distributing software is via *packages*.
- A package provides additional functionality, usually for a special-purpose analysis.
- CRAN is the major repository for general-purpose packages. Bioconductor is the repository for packages focusing on analysis of high-throughput biological data.
- You can use `biocLite` (details later), `install.packages` and `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. Simply unpacking the archive *does not work*.

Packages - Bioconductor

- The most reliable way to install Bioconductor packages (and their dependencies) is to use `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.

Installing versus Loading Packages

- `biocLite` or `install.packages` retrieves a package from a repository, and installs the package in an appropriate location.

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("Biobase")
```

- A package is installed only once.
- During an R session, use `library` to load a package in order to obtain access to its functionality.

```
> library(Biobase)
```

- A package is loaded only once per session.

Name spaces

- When packages are loaded into R, they are attached to the `search` list, see `search`.
- This creates the possibility of symbol masking: the same name being used for different functions in different packages.
- Use, e.g., `stats::runmed` to specify a variable defined in a particular package.
- Many packages have name spaces to help the developer avoid symbol masking.
- A package with a name space may define symbols that are not on the search path. Use the `:::` to access these symbols.

Control-Flow

R has a standard set of control flow functions:

- Looping: `for`, `while` and `repeat`.
- Conditional evaluation: `if` and `switch`.

Examples of Useful String Functions

```
> as.roman(1:10)
 [1] I    II   III  IV   V    VI   VII  VIII IX   X
> paste("chr", as.roman(1:5), sep="")
[1] "chrI"   "chrII"  "chrIII" "chrIV"  "chrV"
> paste("chr", as.roman(1:5), sep="", collapse=", ")
[1] "chrI, chrII, chrIII, chrIV, chrV"
> grep("IV", as.roman(1:30))
[1]  4 14 24
> grep("I[VX]", as.roman(1:30), value=TRUE)
[1] "IV"   "IX"   "XIV"  "XIX"  "XXIV" "XXIX"
```

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.

```
> a = matrix(runif(1e6), ncol=10)
> ## 'apply'
> s1 = apply(a, 1, sum)
> ## 'for', pre-allocating for efficiency
> s2 = numeric(nrow(a))
> for(i in 1:nrow(a))
+   s2[i] = sum(a[i,])
> ## purpose-built function (must faster!)
> s3 = rowSums(a)
```

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 (discussed later), 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" or "objects") of application-oriented concepts and to write functions ("methods") that operate on these objects.
- The R language has two different object oriented paradigms. S3 is used extensively in CRAN packages. S4 is used in Bioconductor and other projects where it is important to represent complex data structures and to interoperate between several packages.
- 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.
- In S4, a class consists of a set of *slots*. Each slot contains a specific type (character, numeric, etc.). Information from slots is usually obtained using "accessor" functions.

Generic Functions and Methods

- A *method* is a function that has been registered, together with the number and classes of its arguments(!) (its "signature"), with a generic.
- A *generic* is a function that examines the classes of its arguments and invokes the most appropriate specific method.
- For instance, the `show` generic is a function used to display the contents of S4 objects. If `show` is called on an instance of class B and there is no *show* method for class B, the show methods for class A will be used.

Classes (S4 example)

```
> setClass("Experiment",
+     representation(assay="matrix", phenotype="data.frame"))

[1] "Experiment"

> setMethod(show, signature(object="Experiment"),
+     function(object)
+ {
+     cat("class: ", class(object), "\n")
+     cat("assay dimensions: ", dim(object@assay), "\n")
+     cat("phenotype names: ", names(object@phenotype), "\n")
+ })

[1] "show"
attr(,"package")
[1] "methods"

> x <- new("Experiment", assay=matrix(rnorm(100), ncol=5),
+          phenotype=data.frame(Gender=sample(c("M", "F"), 5, TRUE),
+                               Age=30 + ceiling(10 * runif(5))))
> show(x)

class:  Experiment
assay dimensions:  20 5
phenotype names:  Gender Age
```

Selected References

- *Software for Data Analysis: Programming with R* by J. Chambers.
- *R Programming for Bioinformatics* by R. Gentleman.
- *Lattice: Multivariate Data Visualization with R* by D. Sarkar.
- *Introductory Statistics with R* by P. Dalgaard.
- *Modern Applied Statistics*, *S Programming* by W. N. Venables and B. D. Ripley.

Course resource

- *Bioconductor Case Studies* by F. Hahne, W. Huber, R. Gentleman, and S. Falcon.