Introduction to R

Nishant Gopalakrishnan, Martin Morgan

Fred Hutchinson Cancer Research Center

9-10 December, 2010

Getting help in R

- `help` and `?`: `help("data.frame")` or ? data.frame
- `help.search("slice")`, `apropos("mean")`
- `browseVignettes("Biobase")`
- `RSiteSearch` (requires internet connection)
- R/Bioconductor mailing lists (`sessionInfo()`)

Data structures in R

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

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

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"
[10] "j" "k" "l" "m" "n" "o" "p" "q" "r"
[19] "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] "" "" "" "" ""
```

Naming vectors

The elements of a vector can be named

- at creation time
- 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
```

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]  10
  ```

- If a zero or `NA` occurs as a subscript in this situation, it is ignored.

Subsetting with negative indices

- 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 indices

- 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
 [7]  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.

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.

Factors

- A special type of vector with grouping information about its components
- A vector with its components grouped with distinct levels
- ```
  > col <- c("red", "green", "red", "yellow", "red")
  > factor(col)
  ```

  ```
  [1] red    green  red    yellow red
  Levels: green red yellow
  ```
  ```

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 dimensions of matrix

```
> 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 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

>
```

Lists

- 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
```

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.

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.

Subsetting lists

- 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.

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.

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).

Accesssing 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 to 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.

Create a data frame

```
> df <- data.frame(type=rep(c("case", "control"), c(2, 3)),
+                  time=rexp(5))
> df
     type      time
1    case 1.1745712
2    case 1.1691266
3 control 0.8227643
4 control 0.1301390
5 control 1.0581316
> df$time
[1] 1.1745712 1.1691266 0.8227643
[4] 0.1301390 1.0581316
```

Update row names

```
> names(df)
[1] "type" "time"
> rn <- paste("id", 1:5, sep="")
> rownames(df) <- rn
> df[1:2, ]
    type     time
id1 case 1.174571
id2 case 1.169127
```

Control Flow

R has a standard set of control flow functions:

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

apply family of functions

- 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 (much 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.

Functions

- 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.

Visualizing data in R

Basic plots

- plot: x-y plotting
- boxplot: box-whisker plot
- hist: histogram
- barplot: bar plot

Basic scatter plot

```
> df <- data.frame("y" = 1:10, "x" = rnorm(10))
> plot(df$x, df$y, col = "red")
```

Trellis graphics

Lattice package

- xyplot: scatter plot
- bwplot: box-whisker plot
- histogram: histogram
- densityplot: kernel density plot

Lattice plots

```
> xyplot( y ~ x | c, data , groups = g)
```

- lattice function
- formula
  - primary variables
  - conditioning variable
- grouping variable
- data

Reading/writing data from/to files

- `read.delim("file")`, `read.table("file")`
- `write.table`, `write`
- `load`, `save`

Packages

- In R the primary mechanism for distributing software is via *packages*.
- The most reliable way to install Bioconductor packages (and their dependencies) is to use `biocLite`.

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

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

```
> library(Biobase)
```

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.