# An introduction to R

Martin Morgan (mailto:mtmorgan@fhcrc.org)
Computational Biology Shared Resource
Fred Hutchinson Cancer Research Center
Seattle, WA, USA

11 September, 2008

## Contents

## 1 Introduction

This document introduces R. It starts with the assumption that the user is new to R, and concludes with the hope that the user is comfortable doing basic data manipulation and analysis, and is equiped with the tools for discovering the power and diversity of R for statistical programing.

This section describes the Computational Biology Shared Resource, and the steps required to install and use R for the first time. The next section walks through a simple work flow reading in data, exploring it, and performing linear regression (in the body of the text) or principle components analysis (in one of the exercises). The third section introduces R as a programming environment. The document concludes with a few 'power tips' that might lead to more productive use of R.

The initial sections are pedantic, but subsequent sections assume increasing familiarity with R's help system and the way R 'works'; in these later sections functions are used without explict discussion of what they are doing.

## 1.1 About the Computational Biology Shared Resource

The Computational Biology Shared Resource is a small (2 member!) service group within Computational Biology. We take on a diversity of roles providing center-wide assistance related to microarray data analysis, high throughput sequencing analysis and tool development, and training in software (especially R and BioConductor) use. We'll engage in 'two-week' projects on an *ad hoc* and *pro bono* basis, and take larger stakes in projects when resources and opportunities for involvement permit.

## 1.2 Installing R

We will install R from a thumb drive, to avoid overwhelming the network. Normally, though, R can be installed from the internet; one can also use R on PHS servers. The starting point for access is `http://cran.r-project.org` and a local mirror of the core of the repository is at `http://cran.fhcrc.org` (the acronym CRAN stands for the Comprehensive R Archive Network). The R web site and local mirror contain pre-compiled binaries with standard installers for MacOS and Windows operating systems, and a number of linux distributions; third party installations using a variety of Linux package managers are also available.

R consists of a core application, required and recommended *packages*, and on some operating systems a graphical user interface. The core application is written primarily in C, released under the GPL, and the source code is readily available (using SVN). Required and recommended packages implement key functionality (e.g., base contains system and other core functionality, stats contains functionality for standard descriptive statistics, probability distributions, random number generators, parametric and non-parametric tests, linear models, time series, factor analysis, and the like).

Interaction with R is through a command-line style interface. The user is presented with a prompt >, types symbol names or expressions, and presses the carriage return to submit the symbols to the R parser and evaluator. R responds with numerical or graphical output. Graphical user interfaces available as part of the basic R installation generally integrate the command-line style interface, graphical output, and help systems; they do not provide a 'point-and-click' solution (other R packages such as Rcmdr attempt this, usually by presenting a subset of R's overall functionality).

A particular strength of R is that functionality provided in the basic installation is augmented by packages contributed by the user community. There are well over 1000 user-contributed packages. Package quality can be excellent, implementing very sophisticated functionality and cutting-edge research methodologies. CRAN provides a comprehensive registry of packages, including 'views' that attempt to group some packages by functionality. Additional projects such as BioConductor (focussing on analysis of high-throughput biological data, with over 250 packages) represent additional resources.

The R web site contains references to books on R,

## 1.3 Exercises: a very first R session

**Exercise 1**
*Copy the OS-specific folder hierarchy from the memory stick to a convenient location on your computer. The following assumes that you copied the hierarchy to a folder named `RIntro` (it avoids confusion in R to use `/` for the file path separator on Windows).*

**Exercise 2**
*Install R.*

1. *Windows and Mac: double click on the installer and follow directions.*
2. *Linux: consult with tutorial assistants.*

*Normally, R can be installed from `http://cran.fhcrc.org`.*

**Exercise 3**
*Check installation / a first R session.*

1. *Start R (e.g., double-click on the appropriate icon, or select from the 'start' menu.*
2. *Enter the text that appears after the `>` and confirm that you are using R version `2.8.0 Under development (unstable)` :*

```
> sessionInfo()

R version 2.8.0 Under development (unstable) (2008-08-29 r46457)
x86_64-unknown-linux-gnu

locale:
LC_CTYPE=en_US.UTF-8;LC_NUMERIC=C;LC_TIME=en_US.UTF-8;LC_COLLATE=en_US.UTF-8;LC_MONETAR

attached base packages:
[1] stats     graphics  utils     datasets  grDevices methods
[7] base
```

3. *Find help about the functions provided by a library with the command*

```
> library(help = stats)
```

4. *Find help about a function, e.g.,* `fivenum` *with the command*

   ```
   > `?`(fivenum)
   ```

5. *Load a special purpose library (in this case, for more advanced plotting)*

   ```
   > library(lattice)
   ```

   *End the R session (select* `n` *when prompted to save the session).*

   ```
   > q()
   ```

## 2 A first work flow

### 2.1 Data input

Start R, and navigate so that files can be easily read. For example,

```
> setwd("~/sharedrsrc/presentations/RIntro")
```

Since \ is used to 'escape' characters in R, it is best to use / for the path separator, even on Windows. To confirm that you've arrived at the intended directory, execute the `list.files` function

```
> list.files()
[1] "extdata"
```

`list.files` is a fairly typical function. View its help page with the command `?list.files`. The usage section indicates how the function can be invoked. There are 6 arguments, all of which are named and have a default value (e.g., the argument named *path* has default value `"."`). Above, we specified no arguments, so R used default values for each. On the other hand, we might have written

```
> list.files("extdata", full.names = TRUE)
```

This illustrates two features of function invocation: unnamed arguments are matched by position (i.e., R assigns `extdata` to *path*), named arguments are matched exactly, regardless of position (i.e., *full.names* is assigned `TRUE`).

The directory `extdata` contains a 'comma-separated value' file, a format exported by many spreadsheets. The file contains three columns of data. The first is the row number. The second and third are plant weight, and a character string representing whether the weights in the same row are from a 'control' or 'treatment' group. Our file has a `header` row, containing the names for the columns, and the row names are in column 1. R has several functions to read in text files. Here we use `read.csv` which, as its name suggests, is specialized for reading comma-separated value files. Our file has a header row, so we indicate this by setting the argument *header* to the logical value `TRUE`. We use the *row.names* function argument to indicate the column containing rows. To discover these arguments and their interpretation, we might have used `?read.csv` to obtain the help page, or, when more confident of the function, `args(read.csv)` to be reminded of the available arguments. We want to read the contents of the file in to R, and to assign the result to a variable that we can reference later. We'll call the variable `weights`, and read the data in with

```
> weights <- read.csv("extdata/weights.csv", header = TRUE,
+     row.names = 1)
```

R automatically adds the '+' sign at the beginning of a line when it continues an incomplete command. You don't need to enter it.

## 2.2 Object exploration and manipulation

We can now manipulate `weights` to find out about it's content. R has a number of different classes of objects, including user-defined classes. We can find out what class `weights` is

```
> class(weights)
```

```
[1] "data.frame"
```

Ah, `weights` is a `data.frame`. A look at `?data.frame` might be good in the long term. For now, a `data.frame` is a matrix-like object where all elements in a row are of the same type, but columns can have different type. We can peek at the top of our `weights`, and summarize its content:

```
> head(weights)
```

```
  Weight Group
1   4.17   Ctl
2   5.58   Ctl
3   5.18   Ctl
4   6.11   Ctl
5   4.50   Ctl
6   4.61   Ctl
```

```
> dim(weights)
```

```
[1] 20  2
```

The display from `head` is a useful confirmation that we have read our data in correctly: there are row names (integer values) followed by two columns of data. `dim` can be used to determine the dimensions of array-like objects; here we see that there are 20 rows and 2 columns.

The function `summary` provides a quick summary of each column of the data frame.

```
> summary(weights)
```

```
     Weight       Group
 Min.   :3.590   Ctl:10
 1st Qu.:4.388   Trt:10
 Median :4.750
 Mean   :4.846
 3rd Qu.:5.218
 Max.   :6.110
```

The Weight column contains numerical values, and the summary is appropriate for this type of data: an indicate of the minimum, first, median, and third quartiles, maximum, and mean of the 20 observations in the column. On the other hand, the summary for Group is different: it indicates that there are 10 entries labeled Ctl, and an equal number labeled Trt.

It turns out that read.csv has read the two columns of data in as different classes. We can determine the class of each column by selecting the column and using class on the result. There are two ways to select a column of a data frame, using $ or [[ Thus:

```
> class(weights$Weight)

[1] "numeric"

> class(weights[["Weight"]])

[1] "numeric"
```

The Weight column contains numeric data. The numeric data type represents real numbers. Other common data numerical types include integer and complex.

Let's look at the column Weight in all its detail

```
> weights[["Weight"]]

 [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14 4.81 4.17
[13] 4.41 3.59 5.87 3.83 6.03 4.89 4.32 4.69
```

The display is of numeric values 4.17, 5.58, etc. For convenience, the index of the numeric value is printed at the start of each line, e.g., [1] indicates the first element.

The column Weights is an example of a vector. All atomic (the meaning of 'atomic' will become apparent later) objects in R are vectors. Most R operations act on vectors. For instance,

```
> log(1 + weights[["Weight"]])

 [1] 1.642873 1.884035 1.821318 1.961502 1.704748 1.724551
 [7] 1.819699 1.710188 1.845300 1.814825 1.759581 1.642873
[13] 1.688249 1.523880 1.927164 1.574846 1.950187 1.773256
[19] 1.671473 1.738710
```

adds 1 to each element of `Weight`, and then takes the natural logarithm of each value. This will be discussed further below.

Returning to our exploration of the `weights` data frame, the `Group` column can be accessed and its class determined in the same fashion as `Weights`:

```
> weights[["Group"]]

 [1] Ctl Ctl Ctl Ctl Ctl Ctl Ctl Ctl Ctl Ctl Trt Trt Trt Trt Trt
[16] Trt Trt Trt Trt Trt
Levels: Ctl Trt

> class(weights[["Group"]])

[1] "factor"
```

R recognized the column of character values in the input file, and interpreted them as factors in the statistical sense. The value of `Group` is again a vector, but this time a vector of factor levels. There are two different levels of the `Group` factor:

```
> levels(weights[["Group"]])

[1] "Ctl" "Trt"
```

These levels are unordered; R also understands ordered factors, and can treat character sequences as just strings without statistical meaning; the *stringsAsFactors* and *colClasses* arguments to `read.csv` influence how data types are read in to R. A factor is not usefully summarized by concepts such as minimum or median, so `summary` provides a different description: a tabulation of the number of observations of each level. A more direct way of obtaining the counts is with the `table` function:

```
> table(weights[["Group"]])

Ctl Trt
 10  10
```

One often wants to operate on (e.g., determine the class of) columns of a data frame (or rows or columns of a matrix). R offers a number of `apply`-like functions that make it convenient to perform the same operation repeatedly. For example, we can 'apply' the `class` function to each column of `weights`, and simplify the result to a vector of characters

```
> sapply(weights, class)

   Weight     Group
"numeric"  "factor"
```

A final useful tool for exploring R objects is `str`, which reveals the internal structure of the object.

```
> str(weights)
```

```
'data.frame':        20 obs. of  2 variables:
 $ Weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
 $ Group : Factor w/ 2 levels "Ctl","Trt": 1 1 1 1 1 1 1 1 1 1 ...
```

We see in the result of this function all of the information we have discovered already, but in a single place and compactly represented: `weights` is a data frame with 20 observations of 2 variables. The first variable, `Weights`, is a numeric ('`num`') vector the first several values of which are presented, and so on. Aspects of the presentation are cryptic at first (e.g., why are the entries for `Group` given as integers?) but are informative as R becomes more familiar (`factor` objects are encoded as integers indexing the corresponding level; this makes important operations on factors compact and efficient).

## 2.3 Visualization

Complex data objects, especially relations between variables, are often best explored through visualization. One way to visualize our data is with the `plot` function, providing 'x' and 'y' variables as arguments, for instance

```
> plot(weights[["Group"]], weights[["Weight"]])
```

Let's take two steps that are more complicated than this. First, we add a *package* to the library of packages available in the current session of R. A package is a collection of R functions and other objects that augment built-in functionality. R starts with a list of packages already loaded. We'll add the lattice package to this list, with the following command:

```
> library(lattice)
```

We can see the set of packages that R searches for functions and other objects with

```
> search()
```

```
 [1] ".GlobalEnv"        "package:lattice"   "package:stats"
 [4] "package:graphics"  "package:utils"     "package:datasets"
 [7] "package:grDevices" "package:methods"   "Autoloads"
[10] "package:base"
```

When the user requests an object, e.g., by invoking a function or attempting to display the contents of a data frame, R searches first in the 'global environment', which is where user objects like `weights` get created by default. If the object is not found in the global environment, R continues to search loaded packages, in the order specified by `search()`, until the object is found. For instance, the function `class` is found in the base package.

We will use `bwplot` from lattice to visualize our data; 'bw' is an abbreviation of box-and-whiskers; loading lattice loads associated help files, so `?bwplot` would

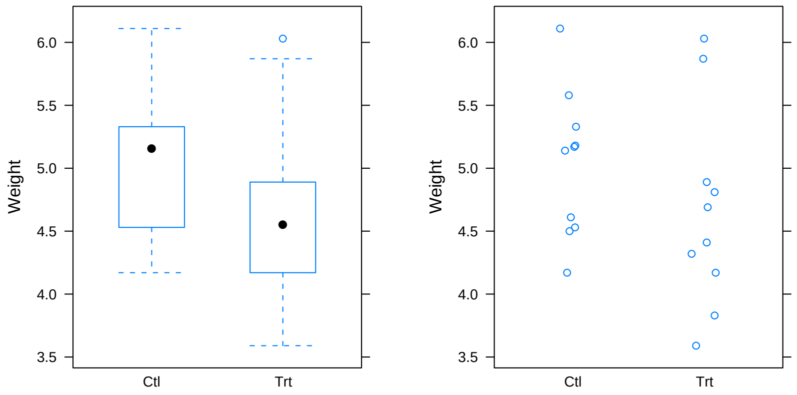

Figure 1: Box-and-whisker and strip plots of `weights`, using the formula `Weight ~ Group`

take us to the help page for this function. The first argument to `bwplot` is `formula`. A `formula` is a notation with a left-hand side and a right-hand side, separated by the ~ character. We could think of visuallizing `weights` as a box-and-whiskers plot where the left-hand side (`Weight`), plotted on the y-axis) is a function of the `Group` to which the `Weight` belongs (plotted on the x-axis). The formula is `Weight ~ Group`. We finish creating the plot by telling `bwplot` where to find the variables specified in the formula (i.e., in the `weights` data frame) and then print the result

```
> print(bwplot(Weight ~ Group, weights))
```

A more detailed look at the data might use `stripplot` with the additional argument `jitter=TRUE` to add small random offsets to the group classification and thus reduce problems distinguishing overlapping data points (not really a problem with the current small data set).

```
> print(stripplot(Weight ~ Group, weights, jitter = TRUE))
```

Results of these displays are in Figure 1.

## 2.4 A little statistics

As an elementary introduction to statistical analysis in R, we can calculate the mean (or other) statistical value of each level in `Group` with another member of the `apply` family mentioned above:

```
> tapply(weights[["Weight"]], weights[["Group"]], mean)
```

```
  Ctl   Trt
5.032 4.661
```

A slightly more elaborate example fits a linear model to our data, treating `Weight` as the dependent and `Group` as the independent variable, using the formula notation introduced above.

```
> lm(Weight ~ Group, weights)

Call:
lm(formula = Weight ~ Group, data = weights)

Coefficients:
(Intercept)     GroupTrt  
      5.032       -0.371  
```

This model implicitly includes the intercept; a slightly different formula removes the intercept

```
> fit <- lm(Weight ~ Group - 1, weights)
> fit

Call:
lm(formula = Weight ~ Group - 1, data = weights)

Coefficients:
GroupCtl  GroupTrt  
   5.032     4.661  
```

The forgoing illustrates that the result of a linear model can be assigned to a variable (as can the result of the lattice plotting functions). That variable can later be interrogated, e.g., to display alternative summaries

```
> summary(fit)

Call:
lm(formula = Weight ~ Group - 1, data = weights)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.0710 -0.4937  0.0685  0.2462  1.3690 

Coefficients:
         Estimate Std. Error t value Pr(>|t|)    
GroupCtl   5.0320     0.2202   22.85 9.55e-15 ***
GroupTrt   4.6610     0.2202   21.16 3.62e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared: 0.9818,	Adjusted R-squared: 0.9798 
F-statistic: 485.1 on 2 and 18 DF,  p-value: < 2.2e-16 
```

```
> anova(fit)

Analysis of Variance Table

Response: Weight
          Df Sum Sq Mean Sq F value    Pr(>F)
Group      2 470.46  235.23  485.05 < 2.2e-16 ***
Residuals 18   8.73    0.48
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

or to add the residuals and fitted values of the linear model to our data frame

```
> weights[["Residuals"]] <- resid(fit)
> weights[["Fitted"]] <- fitted(fit)
> head(weights)

  Weight Group Residuals Fitted
1   4.17   Ctl    -0.862  5.032
2   5.58   Ctl     0.548  5.032
3   5.18   Ctl     0.148  5.032
4   6.11   Ctl     1.078  5.032
5   4.50   Ctl    -0.532  5.032
6   4.61   Ctl    -0.422  5.032
```

The forgoing is meant as a glimpse into the statistical functionality provided by R; statistical capabilities are much more extensive.

## 2.5 Ending the session

Suppose we have accomplished the tasks we set out to do, and it is now time to end the R session. One might be interested in saving particular objects, so that they can be easily recovered at a later session. To save the `weights` and `fit` objects to a single file `weightsAndFits.rda` (the extension `rda` is a convention for files containing R objects saved in the R internal representation) in the (existing) directory `data` , one would evaluate, e.g.

```
> save(weights, fit, file = file.path("data", "weightsAndFits.rda"))
```

These objects could be read in to another R session with

```
> load(file = file.path("data", "weightsAndFits.rda"))
```

A different approach is to save the entire session, so that the session can be restarted at the point where the analysis ended. This is done when quitting R, e.g.,

```
> q(save = "yes")
```

By default, the session is saved in the current working directory as a file named `.RData`. R searches for an `.RData` file in the directory in which R starts, and so would read in and reestablish the session on startup (provided R is started from a directory where the `.Rdata` file will be found).

Experienced R users rarely seem to use the *save="yes"* argument to `quit`. One reason is that the `.RData` file read in depends on the directory used to start R, and the user is surprised to find unexpected (if `.RData` was read in on startup, contrary to user expectation) or missing (if `.RData` was not read in) values in their environment. Instead, the usual practice, especially for small or computationally inexpensive analyses, is to write a script file `MyScript.R` and use the `source` function to re-evaluate the script in a subsequent R session. The script for the session we have just completed is in the file `doc/IntroRLecture.R`.

## 2.6 Exercises

**Exercise 4**
*Repeat the analysis in the text. Specifically:*

1. *Use `setwd` to navigate to the `extdata` directory.*
2. *Use `read.csv` to read the file `weights.csv` into an R object `weigths`.*
3. *Use `head` to 'peek' at the data, `summary` to summarize the columns, and the functions `sapply` and `class` to determine the class of each column.*
4. *Use `library` to load the package lattice to the library of packages available in your R session.*
5. *Use `bwplot` and `xyplot` to visualize the relationship between `Weight` and `Group`.*
6. *Use `lm`, `summary`, and `anova` to explore different formulations of a simple liinear model relating `Weight` to `Group`.*

See the text for solutions to this exercise.

**Exercise 5**
*The file `extdata/weights-table.txt` contains the weights data in another typical format, with two tab-delimited columns. Here are the first five lines of the file:*

```
Weight      Group
4.17       Ctl
5.58       Ctl
5.18       Ctl
6.11       Ctl
```

*Read this data in to R and verify that the results of `summary` and the column classes are the same as in the previous exercise. The file is tab-delimited, so use the function `read.delim` consulting its help page (`?read.delim`) as necessary.*

```
> wtsTable <- read.delim("extdata/weights-table.txt")
> head(wtsTable)

  Weight Group
1   4.17   Ctl
2   5.58   Ctl
3   5.18   Ctl
4   6.11   Ctl
5   4.50   Ctl
6   4.61   Ctl

> summary(wtsTable)

     Weight      Group
 Min.   :3.590   Ctl:10
 1st Qu.:4.388   Trt:10
 Median :4.750
 Mean   :4.846
 3rd Qu.:5.218
 Max.   :6.110
```

**Exercise 6**
*Both* `read.csv` *and* `read.delim` *are 'wrappers' around the function* `read.table`*. The wrappers are designed to make input of particular formats easy. How must you invoke* `read.delim` *to successfully input the csv-delimited data? The tab-delimited data?*

```
> df1 <- read.table("extdata/weights.csv", header = TRUE,
+     sep = ",", row.names = 1)
> df2 <- read.table("extdata/weights-table.txt", header = TRUE)
```

**Exercise 7**
*R packages contain many 'built-in' data sets. Read in Fisher's 'iris' data using the* `data` *command*

```
> data(iris)
```

*Read a brief description of the* `iris` *data set by consulting its help page with the command* `?iris`*.*

1. *Get a feeling for the data with* `head` *and* `summary`*. How many variables and observations are there? What variables are present? What are their classes?*

2. *Use `splom(iris)` to view a conditional scatter plot matrix of the data. Use the argument pch to change the plot character used to represent points, e.g., as solid circles (`pch=20`) or as small dots (`pch="."`). Available plot characters are documented on the `?points` page.*

3. *Use `xyplot` with the formula `Sepal.Width ~ Petal.Width` and the `iris` data frame to visualize the relationship between sepal and petal widths. Using the same formula, add the argument `group=Species`.*

4. *The `princomp` function performs principle components analysis. One way of invoking this function is to provide a formula and a data frame as arguments. The formula has the form `~ x + y + z`, where the left-hand side of the formula is empty and the right hand side of the formula is an expression containing the names of the columns of the data frame that are to be used in the analysis. Using these hints, construct a formula to perform principle components analysis of the variables `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width`. Use this formula and the iris data set as arguments to perform a principle components analysis; assign the results to a new variable, e.g., `ipc`. View the results, and use `summary` to view a summary. Consult `?princomp` for some additional ideas.*

Data exploration might proceed as:

```
> data(iris)
> head(iris)
> summary(iris)
> splom(iris, pch = 20)
> xyplot(Sepal.Width ~ Petal.Width, iris, group = Species)
```

A principle components analysis might start with:

```
> ipc <- princomp(~Sepal.Length + Sepal.Width + Petal.Length +
+     Petal.Width, iris)
> summary(ipc)
```

```
Importance of components:
                          Comp.1     Comp.2     Comp.3
Standard deviation     2.0494032 0.49097143 0.27872586
Proportion of Variance 0.9246187 0.05306648 0.01710261
Cumulative Proportion  0.9246187 0.97768521 0.99478782
                            Comp.4
Standard deviation     0.153870700
Proportion of Variance 0.005212184
Cumulative Proportion  1.000000000
```

# 3 The R programming language

This section highlights a few key features of the R programming language. Additional detail can be found in the `R_intro_lecture.pdf` file distributed with these course notes, and in An Introduction to R.

## 3.1 Basic data types

Basic data types in R are *atomic* vectors. Atomic vectors include `logical`, `integer`, `numeric`, `complex`, or `character` types. Atomic vectors can be created with calls to 'constructor' functions such as

```
> numeric(5)
```

```
[1] 0 0 0 0 0
```

which creates a numeric vector of length 5, initialized with all elements equal to 0. Additional constructors are useful, for example constructing a range of integer values

```
> 1:5
```

```
[1] 1 2 3 4 5
```

```
> seq(1, 10, by = 2)
```

```
[1] 1 3 5 7 9
```

The `c` function concatentates specific values into a vector, with implict coercion to the type required to represent the elements as an atomic vector.

```
> c(1, 2, 3)    # numeric (i.e., real) vector of length 3
```

```
[1] 1 2 3
```

```
> c(1L, 2L, 3L) # integer vector, 'L' used to specify integer type
```

```
[1] 1 2 3
```

```
> c(1L, 2L, 3)  # coerced to common type: numeric vector
```

```
[1] 1 2 3
```

```
> c(12, "A")    # coercion to character vector
```

```
[1] "12" "A"
```

Repeating elements can be constructed with `rep`

```
> rep(c(TRUE, FALSE), each = 3)
```

```
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

R has useful pre-defined variables, e.g.,

```
> letters
```

```
 [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"
```

```
> pi
```

```
[1] 3.141593
```

Atomic vectors support standard programing concepts of `Inf` and `NaN`, but also the statistical concept of `NA`

```
> x <- c(1, Inf, -Inf, NaN, NA)
> x
```

```
[1]    1  Inf -Inf  NaN   NA
```

```
> typeof(x)
```

```
[1] "double"
```

```
> typeof(c(1L, NA))
```

```
[1] "integer"
```

The elements of atomic vectors can be named,

```
> x <- c(a = 1, b = 2)
> x
```

```
a b
1 2
```

```
> names(x)
```

```
[1] "a" "b"
```

This can add very useful structure to simple data, and as illustrated below can facilitate element extraction. Coercion between types is often implict (e.g., `c(1L, 2L, 3)` is coerced to type `numeric`), but can be made explicit.

```
> as.integer(c(1.41, 3.14))
```

```
[1] 1 3
```

R supports higher dimensional matrices (2-dimensional) and arrays (2 or more dimensions). All elements of a matrix or array must be of the same atomic type. A matrix can be constructed in many ways, but one way that makes R's internal representation apparent is to start with a vector and specify dimensions, e.g.,

```
> matrix(1:12, nrow = 3)

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Notice that the matrix is in *column-major* order; in fact, R represents the matrix as a vector, with addition `attributes` describing how the vector is to be interpretted.

```
> attributes(matrix(1:12, nrow = 3))

$dim
[1] 3 4
```

Row and column elements of matricies (and arrays) can be named, as can the dimension itself

```
> matrix(1:12, nrow = 3, dimnames = list(MyRows = LETTERS[1:3],
+     MyCols = letters[1:4]))

      MyCols
MyRows a b c  d
     A 1 4 7 10
     B 2 5 8 11
     C 3 6 9 12
```

This example uses a `list`. A list is a potentially heterogenous collection of elements. The elements may be atomic or otherwise, including other lists. List elements can be named.

```
> list(alpha = letters[1:4], ints = 1:4, m = matrix(1:12,
+     nrow = 3))

$alpha
[1] "a" "b" "c" "d"

$ints
[1] 1 2 3 4

$m
     [,1] [,2] [,3] [,4]
```

```
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

We have already had extensive contact with the `data.frame`, which is a special type of list. A data frame is a list, with the restriction that each element of the list must be an atomic type, and that all elements must be the same length. A data frame is constructed as, for instance,

```
> data.frame(alpha = letters[1:4], ints = 1:4)
```

```
  alpha ints
1     a    1
2     b    2
3     c    3
4     d    4
```

An `environment` is like a list, in that it can store heterogenous collections of values. Unlike lists, all elements of an environment must be named. And as will become apparent below, environments are almost unique amongst R data structures in providing *pass by reference* semantics rather than *pass by value*. An environment is one way to represent an efficient hash table.

A final and perhaps surprising data type on our tour is the `function`. Users create a function by providing an argument list and body consisting of lines of valid R code. A function returns the value of the last line it executes, so no explicit `return` statement is needed. Users typically assign functions to variables that can then be manipulated by other variables. Here is a simple function that takes one argument, $x$, and squares it. The squared value is the last line (and the first!) of the function that is executed, and is thus the value returned by the function.

```
> square <- function(x) {
+     x * x
+ }
```

We'll see that arithemtic operations are *vectorized*, so our `square` function works on vectors too.

```
> square(1:4)
```

```
[1]  1  4  9 16
```

Notice that our `square` is as much a function as any other function in R, and so can be used in, for instance, other functions like `sapply` (here, `sapply` is taking each element of its first argument, and applying the function in its second argument to the element; what happens in the second line, below?).

```
> sapply(list(1:4, 3:1), square)
```

```
[[1]]
[1]  1  4  9 16

[[2]]
[1] 9 4 1
```

```
> sapply(sapply(list(1:4, 3:1), square), sum)
```

```
[1] 30 14
```

## 3.2 Subsetting

One of the most common operations associated with atomic and other objects is subsetting. We have already seen this to some extent, for instance in selecting the first four entries of the character variable `letters`. For any atomic vector we can subset using positive integers, in any order, to select the corresponding element. We can also select outside the range of the vector index to extend the vector (though this is not usually a good idea). Negative indices remove the corresponding entries; positive and negative indices cannot be used in the same step.

```
> letters[c(2:3, 15:17, 18:16)]
```

```
[1] "b" "c" "o" "p" "q" "r" "q" "p"
```

```
> letters[25:28]
```

```
[1] "y" "z" NA  NA
```

```
> letters[-c(1:20)]
```

```
[1] "u" "v" "w" "x" "y" "z"
```

When atomic types or lists contain named elements, the name can be used to retrieve a specific element.

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

```
a c
1 3
```

Logical vectors can be used for subsetting, and have the useful property that they are *recycled* to match the length of the object they are subsetting. So for a vector of length 10 we can choose every third element with

```
> x <- 1:10
> x[c(FALSE, FALSE, TRUE)]
```

```
[1] 3 6 9
```

We have seen, e.g., in `square`, that arithmetic operations are vectorized. So one way of chosing values of a vector that are, say, divisible by 3 might be to construct a logical index from the appropriate comparison, e.g., using the modulus operator `%%`

```
> x <- c(1, 3, 6:9)
> idx <- x%%3 == 0
> idx

[1] FALSE  TRUE  TRUE FALSE FALSE  TRUE

> x[idx]

[1] 3 6 9
```

This would often be abbreviated into the one-liner `x[x %% 3 == 0]`.

Subsetting extends to matrices and arrays, with a two-dimension comma-separated subscript (for a matrix) replacing the single subscript for a vector. A subscript can be missing, in which case all the corresponding elements of that dimension (all rows, for instance) are selected. Dimensions can be subset differently, e.g., rows subset by negative integer values, columns by logical values or names.

```
> m <- matrix(1:12, nrow = 3)
> m[1:2, -3]

     [,1] [,2] [,3]
[1,]    1    4   10
[2,]    2    5   11
```

Selecting a single row or column of a matrix will return a vector, unless the optional argument *drop* is set to `FALSE`

```
> m[1, ]

[1]  1  4  7 10

> m[1, , drop = FALSE]

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
```

Elements of vectors or matrices can be replaced using subsetting on the left side of the assignment operator.

```
> x <- 1:3
> x[2] <- 4
> x
```

```
[1] 1 4 3
```

Note that R usually has *pass by value* semantics. This means that changing one object does not change any copies of that object:

```
> x <- 1:3
> y <- x    # x and y 'the same', i.e., numerically equal
> x[2] <- 4 # change x
> x         # x changed

[1] 1 4 3

> y         # not y

[1] 1 2 3
```

A final subsetting operation involves the ’double-subset’, `[[`. The distinction between this and the single subset operator `[` is primarily apparent with lists and environments. With a list, the single subset returns a list containing the specified elements. With the double subset, only a single element can be specified, and the result is the element itself and not a list. In the following, the first operation returns a list containing a matrix, the double subset returns the matrix itself.

```
> l <- list(m = matrix(1:12, nrow = 3), n = 1:5)
> l

$m
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

$n
[1] 1 2 3 4 5

> l["m"]

$m
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> l[["m"]]

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Environments can only be subset using the double subset operator, and must be subset using a character vector to identify a named element. Assignment using double subsetting is like that for single subsetting on lists; environments have *pass by reference* semantics and assignment has the *side effect* of modifying any copy of the environment. This is generally surprising behavior to the user, so environments are used only in specific circumstances.

```
> env <- new.env()  # create a new environment, see ?new.env
> env[["x"]] <- 1   # create an element "x", and assign value 1 to it
> env[["x"]]

[1] 1

> env_copy <- env   # make a copy of the environment
> env[["x"]] <- 2   # change the original
> env[["x"]]        # env modified

[1] 2

> env_copy[["x"]]   # but so is env_copy!

[1] 2
```

## 3.3 Useful programming operations

R supports a complete range of built-in operations for manipulating numerical (e.g., `+, sqrt, abs, exp, cos`), logical (e.g., `!` for logical negation, `any` to test whether any element of a logical vector is `TRUE`), and character (e.g., `nchar, gsub, substr`) data. R has additional functions useful for statistical analysis (e.g., `rnorm` for generating normal random deviates; `combn` for generating all combinations of elements, `intersect` to calculate the intersection of two vectors). A distinct feature of R is that many of these operations are *vectorized*, so that 'looping through a vector' is often unnecessary (and very inefficient). Arithmetic operations use recycling, so that in the first line the vector `1` is (conceptually) expanded to a vector of length five, and then added element-wise to the square root of the numbers 1 through 5.

```
> log(1 + sqrt(1:5))

[1] 0.6931472 0.8813736 1.0050525 1.0986123 1.1743590

> dev <- rnorm(100)
> max(dev)

[1] 2.486159

> any(dev > 2)

[1] TRUE
```

```
> sum(dev > 2)
```

```
[1] 1
```

```
> s <- c("Fred", "Frank")
> sub("F", "G", s)
```

```
[1] "Gred"  "Grank"
```

Specifics of these functions are found on their help pages, e.g., `?any`.

R supports all common programming constructs. For instance, conditional evaluation occurs with `if`:

```
> dev <- rnorm(100)
> if (any(dev > 2)) {
+     cat("some deviates greater than 2\n")
+ } else {
+     cat("hmm, all deviates less than 2\n")
+ }
```

```
some deviates greater than 2
```

Braces are used to group multiple lines of code into a single expression.

Iteration over vectors uses `for`:

```
> for (i in 2:1) {
+     cat("I am", i, "\n")
+ }
```

```
I am 2
I am 1
```

```
> lst <- list(a = 1, b = 2)
> for (elt in lst) {
+     cat("I am", elt, "\n")
+ }
```

```
I am 1
I am 2
```

The second example shows that iteration is over elements of atomic vectors. When a `for` loop is used to assign elements to a vector, it is efficient to 'pre-allocate'

```
> results <- numeric(10000)
> for (i in seq_along(results)) {
+     results[[i]] <- aFancyCalculation()
+ }
```

Conversely, `sapply` or `lapply` is often a better choice when the task is to iterate over a set of values, e.g.,

```
> cls1 <- sapply(iris, class)
```

instead of

```
> cls2 <- character(ncol(iris))
> for (i in seq_along(cls2)) {
+     cls2[[i]] <- class(iris[[i]])
+ }
> names(cls2) <- names(iris)
> identical(cls1, cls2)
```

```
[1] TRUE
```

More detail about R syntax can be found with `?Syntax`. Details on programming language constructs can be found with `?if`, etc. The manual An Introduction to R is a useful starting point.

## 3.4 Exercises

**Exercise 8**

*This exercise develops familiarity with subsetting and R's vectorized evaluation.*

1. *Define a variable* `x` *that contains the integers -5 through 5. Evaluate* `x * x`*. Note the length and value of the result, i.e., that the evaluation calculated the square of each element of* `x`*.*

2. *What's the result of evalutating* `sum(x*x)`*?* `sqrt(x)`*?* `x %*% x`*?*

3. *Subset* `x` *to display the sixth value. Now use the subset operator on the left-hand side of the assignment operator to replace the sixth value with* `NA`*. What is the result of* `x * x`*?* `sum(x * x)`*?*

4. *Consult the help page for* `sum`*, especially the argument na.rm. Can you arrange for a variant of* `sum(x * x)` *to return a non-NA value?*

5. *Use* `is.na` *to determine which values of* `x` *are NA. Use* `is.na`*, logical negation (*`!`*) and the subsetting and assignment operators to create a new variable* `y` *containing only non-NA values of* `x`*. Can you remove values of* `x` *that are either NA or whose square is greater than 10? (hint:* `|` *is the logical* `or` *operator applied to vectors).*

```
> x <- -5:5
> x * x
```

```
 [1] 25 16  9  4  1  0  1  4  9 16 25
```

```
> sum(x * x)

[1] 110

> sqrt(x)

 [1]      NaN      NaN      NaN      NaN      NaN 0.000000
 [7] 1.000000 1.414214 1.732051 2.000000 2.236068

> x %*% x

     [,1]
[1,]  110

> length(x * x)

[1] 11

> x[6]

[1] 0

> x[6] <- NA
> x * x

 [1] 25 16  9  4  1 NA  1  4  9 16 25

> sum(x * x)

[1] NA

> sum(x * x, na.rm = TRUE)

[1] 110

> is.na(x)

 [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[11] FALSE

> y <- x[!is.na(x)]
> x[!is.na(x) | (x * x) <= 10]

 [1] -5 -4 -3 -2 -1 NA  1  2  3  4  5
```

**Exercise 9**
*This exercise explores functions and program constructs in a more detail.*

1. *We wrote the function* `square`*, defined as*

```
> square <- function(x) {
+     x * x
+ }
```

*Use this as a template to write a function `sumsq` that returns the sum of squares of its argument. Test this with some vectors of your chosing.*

2. *Modify `sumsq` to take a second argument, na.rm. Have this argument take on a default logical value, say `FALSE`. Pass this argument through to the `sum` function inside `sumsq`. Is this function more or less flexible than your previous version?*

3. *Taking a peek at the solution offered for the previous exercise, and remembering that function arguments can be matched by name or, failing that, by position, can you reason why `sumsq(na.rm=TRUE, c(1, 2, NA))` works?*

4. *What happens if you try to use `sumsq` on a character vector? Can you modify the body of `sumsq` to use `is.numeric` to test whether x is numeric, and if not to issue an error, using `stop`, indicating the class of the argument provided, and the class expected?*

5. *What does R think the sum of squares of a zero-length numeric vector is? What do you think it is?*

```
> sumsq <- function(x) {
+     sum(x * x)
+ }
> sumsq((-5):5)

[1] 110

> sumsq(c(1, 2, NA))

[1] NA

> sumsq <- function(x, na.rm = FALSE) {
+     sum(x * x, na.rm = na.rm)
+ }
> sumsq(c(1, 2, NA))

[1] NA

> sumsq(c(1, 2, NA), na.rm = TRUE)

[1] 5

> sumsq(c(1, 2, NA), na.rm = FALSE)
```

```
[1] NA

> sumsq(na.rm = TRUE, c(1, 2, NA))

[1] 5

> try(sumsq(letters))
> sumsq <- function(x, na.rm = FALSE) {
+     if (!is.numeric(x)) {
+         stop("'x' is '", class(x), "' but should be 'numeric'")
+     }
+     sum(x * x, na.rm = na.rm)
+ }
> try(sumsq(letters))
> sumsq(numeric(0))

[1] 0
```

## 4 Power tips

Effectively using R requires developing a working enviroment that allows you to effectively edit and re-submit commands without excessive typing. One strategy is to open and edit R files (typically ending with `.R`) in a text editor, and transfering code chunks (e.g., via cut and paste) into an interactive R session. For this to work, one would really like an editor that knows about R (or perhaps C) syntax, so that indentation, keyword coloring, and even function lookup are available. Those comfortable with emacs will find ESS (emacs speaks statistics) a very effective tool.

A typical analysis might start with an empty file, e.g., `script.R`. The steps of the analysis might be worked out through interaction with R, with the final version of each step forming a few lines of `script.R`. When complete, the analysis will be contained in `script.R`, and can be performed in its entirety using a command like

```
> source("script.R")
```

or, from the command line,

```
R CMD BATCH script.R
```

Much of the power and flexibility of R comes from it very rich set of functions. Becoming familiar with these functions involves reading introductory documentation (like An Introduction to R), frequently consulting help pages, exploring available packages on CRAN (see `?install.packages` for instructions on how to install additional packages), and seeking help from knowledgable R users. The Hutch has a particularly rich set of R experts, in the form of the Computational

Biology Shared Resource and more generally members of the Computational Biology program involved in the BioConductor project. The R-help mailing list is invaluable both as an archive of previous questions and a resource for getting usually helpful, friendly, and accurate advice.