Introduction to R

Patrick Breheny

January 17, 2013

Our goal for today is to introduce R, an open-source computing language designed to allow for fluid, interactive data manipulation, analysis, and visualization. R is installed on all the computers in the lab, but if you are interested in installing it at home, go to www.r-project.org. You can run R directly, but it is often more convenient to run R through an integrated development environment such as RStudio www.rstudio.com, which is also installed on the lab machines.

Much of the material here is adapted from *S Programming*, by Venables and Ripley (2000), an excellent book on the details of the S and R languages that goes into far more detail than I do here.

1 R objects

Commands in R are either *expressions*, which are evaluated and printed, or *assignments*, which store the result of an evaluation as an object. Arithmetic operations for the most part work very similar to any calculator (note that # marks the rest of the line as a comment):

```
(5^2) * (10 - 8)/3 + 1  ## An expression

## [1] 17.67

x <- (5^2) * (10 - 8)/3 + 1  ## An assignment
x

## [1] 17.67
```

Note that the value of the expression is now stored in an *object* called x. This allows us to use it again in further calculations:

```
x + 1

## [1] 18.67

n <- 50
x/n

## [1] 0.3533
```

Objects can be named using any combination of upper- and lower-case letters, digits 0-9 (provided they are not in the initial position), and the period. Note that R is case sensitive (x and X refer to two different objects).

All objects in R have a *class*, which describes the kind of thing that is stored in the object. For instance,

```
class(x)

## [1] "numeric"
```

tells us that `x` is storing a numeric object at the moment.

1.1 Functions

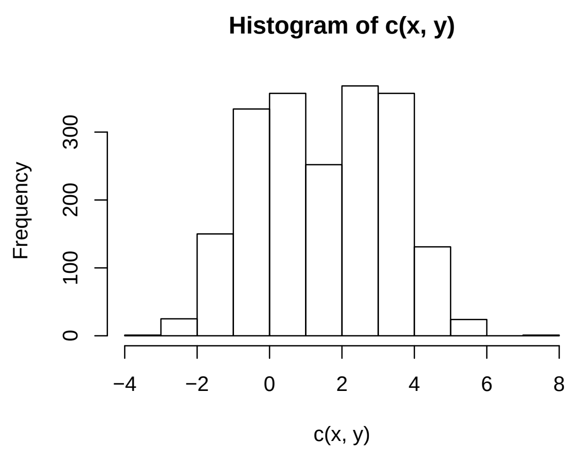
`R` is said to be a functional language, meaning that it is built around calling functions to accomplish tasks:

```
x <- 1:9  ## Creates a vector of numbers 1, 2, ..., 9
mean(x)

## [1] 5

median(x)

## [1] 5

sd(x)

## [1] 2.739

min(x)

## [1] 1

sum(x)

## [1] 45

x^2

## [1]  1  4  9 16 25 36 49 64 81

sum(x^2)

## [1] 285
```

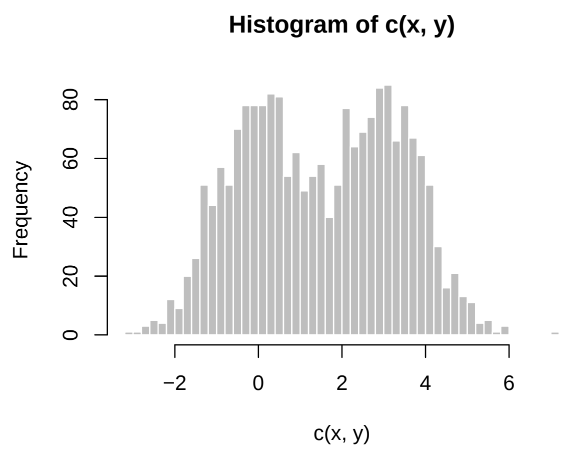
To get more information about any function in `R`, just type `help('sd')` or, more compactly, `?sd`. To search the help files for pages mentioning, say, regression, type `help.search("regression")` or `??regression`. Over time in this course, we'll see a number of functions in `R` and how they are used.

Functions typically have a number of options which may either be specified or left to their default values:

```
x <- rnorm(1000)
y <- rnorm(1000, mean = 3)
hist(c(x, y))
```



```
hist(c(x, y), col = "gray", border = "white", breaks = 40)
```



1.2 Vectors and matrices

The simplest way to create a vector is with `c` (for concatenate). Vectors are typically either numeric, character, or logical:

```
x1 <- c(-0.1, 0.3, -0.9, 0.2, -0.6, -2.3)
x2 <- c("red", "blue", "green", "purple")
x3 <- x1 > 0
x3

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

The elements of a vector may be named and then accessed by name:

```
names(x1) <- c("Justine", "Rachel", "Meng", "Xiuhua", "Whitney", "Paul")
x1
```

```
## Justine  Rachel    Meng  Xiuhua Whitney    Paul
##    -0.1     0.3    -0.9     0.2    -0.6    -2.3

names(x1)

## [1] "Justine" "Rachel"  "Meng"    "Xiuhua"  "Whitney" "Paul"

x1["Justine"]

## Justine
##    -0.1
```

If a vector is arranged into a regular series of rows and columns, it becomes a matrix:

```
X <- matrix(x1, nrow = 2)
X

##      [,1] [,2] [,3]
## [1,] -0.1 -0.9 -0.6
## [2,]  0.3  0.2 -2.3
```

By default, matrices are filled in by column, but this can be changed:

```
matrix(x1, nrow = 2, byrow = TRUE)

##      [,1] [,2] [,3]
## [1,] -0.1  0.3 -0.9
## [2,]  0.2 -0.6 -2.3
```

The rows and columns of a matrix can be named using `rownames` and `colnames`. Individual elements of matrices can be accessed with `X[i, j]`. Arrays further extend this concept, and can have an arbitrary number of additional dimensions (`A[i, j, k]`). The dimensions of a matrix or array can be found using the `dim` function.

The functions `cbind` and `rbind` can be used to join together vectors or matrices column-wise or row-wise:

```
X1 <- matrix(rnorm(6), ncol = 3)
X2 <- matrix(rnorm(6), ncol = 3)
rbind(X1, X2)

##         [,1]    [,2]    [,3]
## [1,]  0.1332  1.2593  0.2757
## [2,] -0.5580  0.7397  0.2725
## [3,]  0.5975  1.7651  2.7919
## [4,]  0.4378 -1.5741 -0.8844

cbind(X1, X2)

##         [,1]   [,2]   [,3]   [,4]   [,5]    [,6]
## [1,]  0.1332 1.2593 0.2757 0.5975  1.765  2.7919
## [2,] -0.5580 0.7397 0.2725 0.4378 -1.574 -0.8844
```

1.3 Lists

Lists are used in R to collect together items of different types. Items in a list may be accessed by number, such as L[[1]], or by name, as in the following example:

```
tt <- t.test(1:7, 5:10)
tt

##
##  Welch Two Sample t-test
##
## data:  1:7 and 5:10
## t = -3.131, df = 10.99, p-value = 0.009576
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.961 -1.039
## sample estimates:
## mean of x mean of y
##       4.0       7.5
##

tt$conf.int

## [1] -5.961 -1.039
## attr(,"conf.level")
## [1] 0.95

tt$p.value

## [1] 0.009576
```

1.4 Data frames

A *data frame* is typically how data sets are stored in R. Like a matrix, a data frame has a regular grid pattern of rows and columns. However, unlike a matrix (instead, like a list), a data frame is not restricted to contain only numeric or only character values. Each column of a data frame has its own type, as is typical for real data (some variables are continuous, others are categorical). Data frames can be constructed directly, but in this class, it will be more typical to read them in from raw, tab-delimited files:

```
tips <- read.delim("http://web.as.uky.edu/statistics/users/pbreheny/760/data/tips.txt")
head(tips)

##   TotBill  Tip Sex Smoker Day  Time Size
## 1   18.29 3.76   M    Yes Sat Night    4
## 2   16.99 1.01   F     No Sun Night    2
## 3   10.34 1.66   M     No Sun Night    3
## 4   21.01 3.50   M     No Sun Night    3
## 5   23.68 3.31   M     No Sun Night    2
## 6   24.59 3.61   F     No Sun Night    4

class(tips$TotBill)

## [1] "numeric"
```

```
class(tips$Sex)

## [1] "factor"
```

Local addresses can be used as well, either relative to the current directory (`getwd`) or as an absolute path. Data frames can have row names in general, although the one above does not. The functions `rbind` and `cbind` can also be used on data frames.

It is often cumbersome to type `tips$` repeatedly to access the elements of the data frame. There are two ways around this: `attach` and `with`. The former is permanent (although it can be undone with `detach` and thus can sometimes lead to unintended side effects, while the latter only acts temporarily:

```
with(tips, mean(TotBill))

## [1] 19.79

mean(TotBill)

## Error:  object 'TotBill' not found

attach(tips)
mean(TotBill)

## [1] 19.79

detach(tips)
```

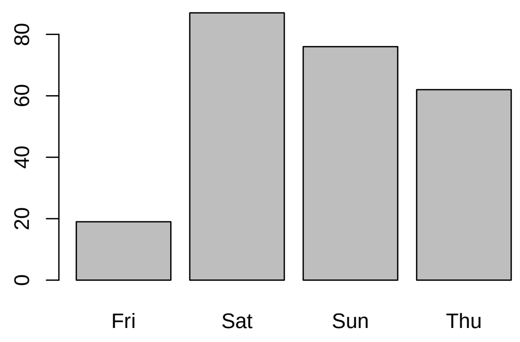
Note that we can add columns to the data frame after it has been created:

```
tips$Rate <- with(tips, Tip/TotBill)
```

1.5 Factors

A factor is a special type of vector used to encode levels of a categorical variable (such as `tips$Sex` above).

```
table(tips$Sex)

##
##    F   M
##   87 157

levels(tips$Sex)

## [1] "F" "M"

barplot(table(tips$Day))
```



2 Indexing

Elements of a vector can be accessed in five distinct ways – it is often easier to use one method in one circumstance and a different method in other cicumstances, so the flexibility `R` provides in this regard is quite convenient:

- A logical vector: Specifies, for each element, whether or not to include it
- A vector of positive integers: Lists the elements to include
- A vector of negative integers: Lists the elements to exclude
- A vector of names: Lists the elements to include by name
- Empty: Select all components

```
x1[x1 > 0]  ## Logical vector

## Rachel Xiuhua
##    0.3    0.2

x1[1:3]  ## Positive integers

## Justine  Rachel    Meng
##    -0.1     0.3    -0.9

x1[-(5:6)]  ## Negative integers

## Justine  Rachel    Meng  Xiuhua
##    -0.1     0.3    -0.9     0.2

x1[c("Whitney", "Meng")]  ## Names

## Whitney    Meng
##    -0.6    -0.9

x1[]

## Justine  Rachel    Meng  Xiuhua Whitney    Paul
##    -0.1     0.3    -0.9     0.2    -0.6    -2.3
```

The last option may seem pointless, but it is necessary among other places when accessing portions of a matrix or data frame (note that we are specifying a subset of rows, but all the columns):

```
tips[tips$Tip >= 7, ]

##     TotBill   Tip Sex Smoker Day  Time Size   Rate
## 25    39.42  7.58   M     No Sat Night    4 0.1923
## 171   50.81 10.00   M    Yes Sat Night    3 0.1968
## 213   48.33  9.00   M     No Sat Night    4 0.1862
```

3 Operations

3.1 Arithmetic

As mentioned above, arithmetic in `R` generally works as you would expect. One wrinkle worth discussing, however, is *recycling*, as illustrated below:

```
x <- sample(1:10, 3)
y <- sample(1:10, 6)
x

## [1] 3 1 9

x + 2  ## Adds 2 to each element of x

## [1]  5  3 11

x + x  ## Adds x to itself elementwise

## [1]  6  2 18

x + y  ## Adds x to y elementwise, 'recycling' x

## [1] 11  7 19  8  3 12
```

If the length of `y` had not been a multiple of the length of `x`, fractional recycling would have occurred, with a warning.

Two important logical operators to know about are `&` (and) and `|` (or):

```
x <- TRUE
y <- FALSE
x & y

## [1] FALSE

x | y

## [1] TRUE
```

There are dozens of useful functions for arithmetic in `R`, most of which have obvious names or can easily be found using `help.search`: `log, exp, sign, sqrt, round, sum, prod, range, sort, intersect`, and many more. Two functions that are perhaps worth mentioning specifically are `seq`, which generates equally spaced sequences of numbers, and `rep`, which is used to repeat an object in various ways:

```
seq(0, 1, 0.2)

## [1] 0.0 0.2 0.4 0.6 0.8 1.0

seq(0, 1, len = 5)

## [1] 0.00 0.25 0.50 0.75 1.00

rep(1, 5)

## [1] 1 1 1 1 1

rep(1:2, 2)

## [1] 1 2 1 2

rep(1:2, c(2, 2))

## [1] 1 1 2 2
```

Finally, it is worth remarking that `R` has a specific value to represent missing data: `NA`. For example, take a look at the data set `airquality` and note that `Ozone` has a number of missing values:

```
sum(is.na(airquality$Ozone))

## [1] 37

mean(airquality$Ozone)

## [1] NA

mean(airquality$Ozone, na.rm = TRUE)

## [1] 42.13
```

Note that `NA` is not the same thing as infinity, and it is not the same thing as ”undefined” – those mathematical concepts have their own representations in `R`:

```
1/0

## [1] Inf

0/0

## [1] NaN
```

3.2 Matrix and array arithmetic

Arrays may be used in ordinary arithmetic the same way as vectors:

```
A <- matrix(rpois(12, 3), nrow = 4)
B <- matrix(rpois(12, 3), nrow = 4)
A * B + 2 * A
```

```
##      [,1] [,2] [,3]
## [1,]   15   14   16
## [2,]   16    9   56
## [3,]   10    3    8
## [4,]   25   10    6
```

Matrix multiplication is a separate operation:

```
t(A) %*% B
```

```
##      [,1] [,2] [,3]
## [1,]   42   34   54
## [2,]   33   20   40
## [3,]   57   22   62
```

```
crossprod(A, B)
```

```
##      [,1] [,2] [,3]
## [1,]   42   34   54
## [2,]   33   20   40
## [3,]   57   22   62
```

```
A %*% B
```

```
## Error:  non-conformable arguments
```

Finally, it is worth knowing that `solve` returns the inverse of a matrix (provided, of course, that it is invertible):

```
X <- crossprod(A)
X.inv <- solve(X)
X %*% X.inv
```

```
##      [,1]       [,2]      [,3]
## [1,]    1 -4.441e-16 6.661e-16
## [2,]    0  1.000e+00 2.442e-15
## [3,]    0 -8.882e-16 1.000e+00
```

3.3 Character operations

`R` also has a number of functions for computing on strings of characters. Perhaps most importantly, `paste`, which pastes together strings:

```
paste(LETTERS[1:4], 1:4)
```

```
## [1] "A 1" "B 2" "C 3" "D 4"
```

```
paste(LETTERS[1:4], 1:4, sep = "")
```

```
## [1] "A1" "B2" "C3" "D4"
```

```
a <- paste(LETTERS[1:4], 1:4, sep = "-")
nchar(a)
```

```
## [1] 3 3 3 3

substr(a, 1, 2)

## [1] "A-" "B-" "C-" "D-"
```

There are also very powerful search, replace, and match functions like `grep`, `gsub`, and `match`, although a detailed discussion of them is beyond the scope of this lecture.

3.4 Vectorized calculations

`R` also offers a number of convenient functions for applying a function to each element of a list, or to each column of a matrix: `apply, tapply, sapply, lapply`.

```
apply(airquality, 2, mean)

##   Ozone Solar.R    Wind    Temp   Month     Day
##      NA      NA   9.958  77.882   6.993  15.804

apply(airquality, 2, mean, na.rm = TRUE)

##   Ozone Solar.R    Wind    Temp   Month     Day
##  42.129 185.932   9.958  77.882   6.993  15.804
```

3.5 Probability distributions

In statistics, we are often interested in working with random numbers and probability distributions. To carry out ”draw n balls from an urn” type of sampling, there is the function `sample`, which can be done with or without replacement:

```
sample(1:10, 5)

## [1] 1 8 3 9 6

sample(LETTERS[1:10], 5)

## [1] "H" "G" "A" "B" "C"

sample(LETTERS[1:5], 10, replace = TRUE)

##  [1] "B" "A" "E" "A" "E" "B" "E" "E" "E" "C"
```

`R` provides a wide array of functions for working with specific probability distributions as well, and they are organized in the following systematic manner (using the normal distribution as an example):

- `dnorm`: The density (*i.e.*, the pdf); for categorical distributions like (*dbinom*), this returns the mass function (pmf)
- `pnorm`: The CDF
- `qnorm`: The quantile function, or inverse CDF
- `rnorm`: Generates random numbers from the distribution

The arguments that each distribution takes are, of course, different, but the organization is the same: there are `dpois`, `ppois`, `qpois`, and `rpois` that allow you to work with the Poisson distribution, and so on for all common distributions.

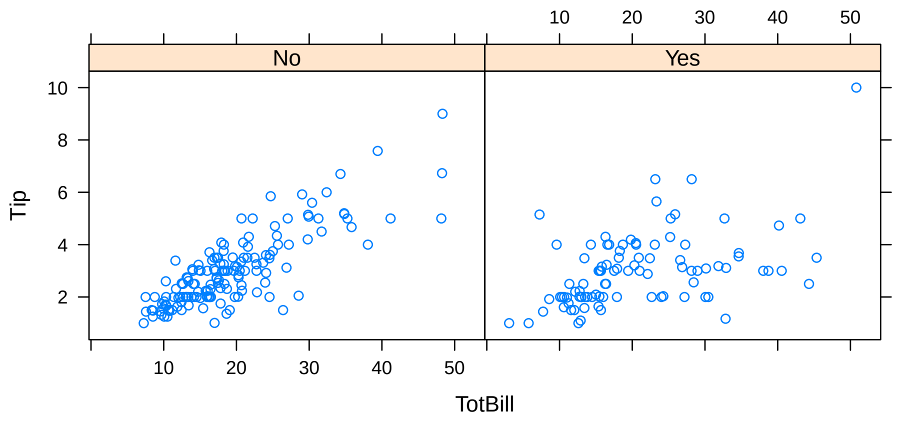
4 Writing your own functions

One of the best things about R (perhaps *the* best thing about R) is how easy it is to write your own functions. For example, let's write a function to solve for the least squares regression coefficients (this function is redundant, of course, as a perfectly good function to do this already exists in R, but it will be a useful exercise regardless).

```
ols <- function(XX, yy) {
    missing.data <- apply(is.na(XX), 1, any) | is.na(yy)
    X <- cbind(Intercept = 1, as.matrix(XX[!missing.data, ]))
    y <- yy[!missing.data]
    solve(crossprod(X)) %*% t(X) %*% y
}
ols(airquality[, -1], airquality$Ozone)

##                 [,1]
## Intercept -64.11632
## Solar.R     0.05027
## Wind       -3.31844
## Temp        1.89579
## Month      -3.03996
## Day         0.27388

## Check against existing R function:
lm(Ozone ~ ., airquality)

##
## Call:
## lm(formula = Ozone ~ ., data = airquality)
##
## Coefficients:
## (Intercept)      Solar.R         Wind         Temp        Month
##    -64.1163       0.0503      -3.3184       1.8958      -3.0400
##         Day
##      0.2739
##
```

The ability to write your own functions greatly enhances one's ability to customize and extend R. In particular, there are hundreds of extra functions that people have written which are available on the Comprehensive R Archive Network (`cran.r-project.org`). These functions are bundled together into *packages* and may be installed with `install.packages` and loaded with `require` or `library`. For example, the `lattice` package (by default installed, but not loaded) provides a number of nice tools for plotting:

```
require(lattice)
xyplot(Tip ~ TotBill | Smoker, data = tips)
```



5 Control structures

One last topic: *control structures* are commands that control whether or not to execute other commands, or whether to repeatedly execute blocks of commands. The `if` statement is used as follows:

```
x <- airquality$Ozone
x <- tips$Sex
if (is.numeric(x)) sqrt(x) else stop("Can't take sqrt of things that aren't numbers")

## Error:  Can't take sqrt of things that aren't numbers
```

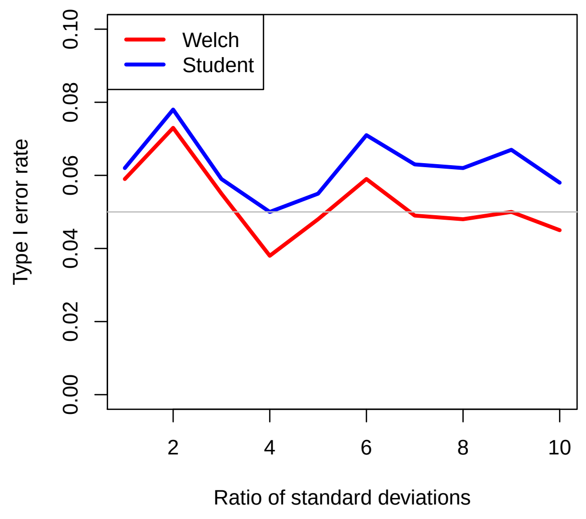
The other basic control structure is `for`, which executes loops:

```
total <- 0
for (i in 1:10) {
    total <- total + i
}
total

## [1] 55
```

6 Example: Running a simulation

To get a sense of how loops used, let's carry out a brief simulation study investigating the robustness of Student's t-test when the variances of the two groups are unequal.

```
N <- 1000
n <- 10
SD <- 1:10
pW <- pS <- matrix(NA, nrow = N, ncol = length(SD))
for (i in 1:length(SD)) {
    print(i)
    for (j in 1:N) {
        s1 <- rnorm(n)
        s2 <- rnorm(n, sd = SD[i])
        pW[j, i] <- t.test(s1, s2, var.equal = FALSE)$p.value
        pS[j, i] <- t.test(s1, s2, var.equal = TRUE)$p.value
    }
}

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

plot(SD, apply(pW < 0.05, 2, mean), type = "l", lwd = 3, col = "red", ylab = "Type I error rate",
    xlab = "Ratio of standard deviations", ylim = c(0, 0.1))
lines(SD, apply(pS < 0.05, 2, mean), lwd = 3, col = "blue")
legend("topleft", col = c("red", "blue"), lwd = 3, legend = c("Welch", "Student"))
abline(h = 0.05, col = "gray")
```



As with any programming language, the only way to learn R is to use R, and certainly, you'll grow more familiar with R and learn many more functions as the semester progresses. Hopefully, this document serves as a useful introduction and reference.