Introduction to R

Patrick R. Stephens

June 9, 2013

1 Introduction

This is the first of three sessions on using the R programming environment for scientific programming. Examples of R code use the `Courier New` font. Topics to be covered today include:

- Basics of R and the RStudio working environment
- Basic operators and functions in R
- R objects including variables, vectors, matrices, data frames, and lists
- Writing functions in R
- Flow control: loops and switches

2 Setting up a workspace using RStudio

RStudio is a user friendly R working environment that makes it easy to organize projects and provides several improvements over the default R working environment such as providing an integrated full featured script editor and remembering all graphics outputs for a session. It is available for most operating systems (windows, mac, and linux), and can be downloaded here: http://www.rstudio.com/

Creating a new project

After opening RStudio:

- under the "Project" tab choose "Create Project."
- From the popup menu choose "New Project."
- Name the new directory (a folder that RStudio will create), and make sure that the folder is being created someplace that you can easily find it.
- Under the "File" tab, choose "New" to open a new script window.
- The folder you created will be used to store all data related to your project including R workspace files, RStudio script files, data files (*e.g.*, csv files), and custom functions. Each independent project that you undertake in R should be organized in a separate folder that contains everything you need for it.

3 Basics of RStudio and the R work environment

The four windows

The script editing window (upper left): This window contains the RStudio script editor. It colors different types of code differently. For example, functions and named objects are colored black, numbers are colored blue, and operators are colored gray. It allows you to execute lines of a script individually or in bulk by selecting the lines of code you wish to execute and hitting the "Run" button near the upper right hand corner of the window, or by hitting "ctrl+enter". Note that when your run any code, it is sent to the R console window to be executed.

The workspace/ history window (upper right): This window has two tabs. Workspace shows you all of the objects currently in memory. History shows you all of the commands that have been executed during a session. Note that the Workspace tab can be used to import data using the Import Dataset button, but this is generally better done manually using commands such as "read.csv" in the script window or the R console.

The R console window (lower left): this is the command line prompt within R. Code can be sent here from the scripting window, or typed or pasted here directly.

Miscellaneous window (lower right): this window has several tabs. Files shows all of the files in the directory you are currently working from. Plots shows the output of all R graphics commands that have been executed during a session. Packages shows what libraries are available and which are currently loaded. Help shows help files that have been queried.

How to run a script/ execute code

When using RStudio, a line of code can be run by typing it directly into the console window, copying it from the scrip editor and pasting it into the console window, or by placing the mouse cursor anywhere in the line and hitting the "run" button in the script editor. Multiple lines of code can be executed by copying and pasting them, or by selecting multiple lines and hitting run.

Tip: When debugging scripts (multiple lines of code that perform some task), it is extremely helpful to run one line at a time to see which lines produce error messages in the R console.

Tip 2: Code that is executed by typing it directly into the R console will not be saved in your script file, but will be saved in the History tab in the workspace/ history window (upper right).

Exercise 1. Type the following lines of code into the RStudio script editor:

```
> print("Hello world.")

[1] "Hello world."

> 4+5

[1] 9

> 4^5

[1] 1024
```

```
> 8*(7+3)

[1] 80
```

Execute each line of code individually, then all at once.

Annotating scripts with

Any line or part of a line of code that is preceded by a # symbol will not be executed by R. You can use this to create notes in your R scripts. Examples:

```
> # R will completely ignore this line of text
> 3+4 #R will execute the first part of this line and ignore the rest

[1] 7

> # if you need to include a long note with several lines of text,
> # be certain that each line is preceded by a #
```

Tip: Annotate well and annotate often. Try to imagine that "future you" has not seen the code you are currently working on for months or years, but wants to use it or modify it for something (this happens a lot more often than you would think). Is it annotated well enough that "future you" can easily figure out what each line of code does?

Exercise 2. Type the following line of code into the script editor and execute it:

```
> 3*4 # This is a comment

[1] 12
```

Saving scripts and workspaces in RStudio

To save the script you are currently editing, simply use "Save" under the "File" tab. RStudio script files are saved with the file name extension ".R" by default. A workspace file is a file that contains everything that is currently in the memory of R (*i.e.*, everything shown under the work space tab). To save the current workspace, under the "Session" tab choose "Save workspace as...". R workspace files are saved with the file name extension ".RData" by default.

Exercise 3. Save your current script.

Creating a "readme" text file

It's a good idea to create a README text file that describes your project and keep it in the project folder. This file should describe the overall project and the files in the project folder. You may also want to describe the objects that are saved in the current workspace, particularly if some of them took a long time to generate. In RStudio this is easy to do. Under the "File" tab choose "New" and then "Text file". Type whatever notes your wish into the text file and then save it using the name "README.txt".

Exercise 4. Create a text file called "README.txt" and save it to your project folder.

Accessing help files

Help files for most functions can be accessed using "?nameofthefunction". The R code of many functions can be accessed by typing the name of the function. Example:

```
> ?cor.test  # this will open up the help file for the "cor.test" function
```

Tip: Most help files have lines of sample code to illustrate some of the basic uses of a function at the bottom of the file. Run this code to get an idea of how a function works, or what it can be used for.

Exercise 5. Access the help file for the hist command.

Packages

R "packages" are libraries of code, specialized collections of functions (and often data), most of which are not part of the core R package. Many of them implement specialized statistical or numerical methods that would not be covered in typical courses. For example, `deSolve` is a package that allows users to numerically solve differential equation models. Differential equation models are often used to represent the dynamics of epidemics, so tools for solving them can be very useful to scientists studying infectious diseases. Packages that are loaded into or are currently active in R are referred to as "attached". In RStudio, new packages can be downloaded and installed by clicking on the "Tools" tab and selecting install packages.

Exercise 6. Download and install the "deSolve" library.

Once installed, packages can be loaded in two ways. One is to use the library command (*i.e.*,"library(nameoflibrary)"). When using RStudio, packages can be loaded and unloaded using the "Packages" tab in the lower right hand window.

```
> library(deSolve)  # this will load the deSolve package
```

A file giving a description of a packages and a list of its authors can usually be opened the same way that other help files are accessed (*i.e.*, "?nameoflibrary"). Example:

```
> ?deSolve   # this accesses the main page of the deSolve help files if deSolve is loaded.
```

Note that at the bottom of any help file associated with a library, there will be a link to the library index. This index contains short descriptions of all of the specialized functions of a library.

Google R style guide

We recommend that you follow the Google R style guide when writing scripts and functions: http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html

I would recommend taking a look at it after working through this introduction to R since much of the advice Google provides won't make sense until you know a bit of R code. Examples in this handout follow style guide recommendations for the most part.

4 Using R as a calculator: operators and basic functions

Arithmetic operators

R has a number of built in arithmetic operators (Table 1). These can be used directly to perform calculations with R, or can be used as part of scripts to define the relationships between variables. Some command line calculation examples:

```
> 4 * 6

[1] 24

> 5 ^ 4

[1] 625

> 119 / 10

[1] 11.9

> 119 %/% 10

[1] 11

> 119 %% 10

[1] 9
```

R code	operation
`x + y`	add x to y
`x - y`	subtract y from x
`x *`	multiple x by y
`x / y`	divide x by y
`x^y`	raise x to the power of y
`x %/% y`	integer division, *e.g.,* `11 %/% 4` is 2
`x %% y`	modulus (remainder after integer division), *e.g.,* `11 %% 4` is 3

Table 1: Arithmetic operators commonly used in R .

The order in which operators are executed follows the rules of basic algebra:

```
> 4 + 2 * 3 + 4

[1] 14

> (4 + 2) * (3 + 4)
```

```
[1] 42
```

Finally, multiple operations need not be on separate lines. They can be separated by semi-colons.

```
> 119/10; 119 %/% 10; 119 %% 10
```

```
[1] 11.9
```

```
[1] 11
```

```
[1] 9
```

Note: it is bad form to execute multiple operations in a single line of code according to the Google R style guide!

Exercise 7. Predict the outcome of the following calculations, and use R to see if you are correct:

1. `4*5`
2. `4 %/% 5`
3. `4 %% 5`
4. `4+3*2`
5. `(4+3)*2`
6. `2^3*2`

Logical operators

R also has many logical operators. These are primarily used in scripts and in functions, but can also be executed at the command line to perform simple true/false checks (this is generally only done to check and see what value a given operator will return given a particular input). Some command line examples:

```
> 4 == 6
```

```
[1] FALSE
```

```
> 6 == 6
```

```
[1] TRUE
```

```
> 4 != 6
```

```
[1] TRUE
```

```
> 4 > 6
```

R code	operation
`x == y`	x equals y
`!`	negation, (*i.e.,* `x!=y` means x does not equal y
`x > y`	x is greater than y
`x < y`	x is less than y
`x >= y`	x is greater than or equal to y
`x <= y`	x is less than or equal to y

Table 2: Logical operators commonly used in R .

```
[1] FALSE
```

Tip: Do not confuse "=" (used to assign objects in R with "==" (the logical operator).

Tip 2: More operators can be found at http://www.statmethods.net/management/operators.html.

Exercise 8. Predict the whether the following statements are true or false, used R to see if you are correct.

1. `4 == 5`
2. `4 >= 5`
3. `2 * 3 != 2 ^ 3 - 2`

Simple Functions

R also has many built in functions that perform simple mathematical operations such as rounding a variable or returning the natural log of a variable. Some command line examples:

```
> floor(3.4)

[1] 3

> ceiling(3.4)

[1] 4

> log(304)

[1] 5.717028

> log10(304)

[1] 2.482874

> abs(44)
```

R code	operation
`floor(x)`	roubd x down to the nearest integer
`ceiling(x)`	round x up to the nearest integer
`round(x)`	round x to its nearest integer
`round(x, digits=n)`	round x to n digits, (*i.e.,* round(3.568, digits=2) returns 3.57)
`log(x)`	natural logarithm of x
`log10(x)`	base 10 logarithm of x
`abs(x)`	absolute value of x
`exp(x)`	the base of the natural logarithm (e) raised to the power of x (antilog of x)

Table 3: Simple built in R functions.

```
[1] 44
```

Tip: more functions can be found at http://www.statmethods.net/management/functions.html .

Exercise 9. Predict the output of the following lines of code calling built in R functions. Use R to see if you are correct.

1. `round(4.3)`
2. `round(4.7)`
3. `round(3.217743, digits = 4)`
4. `exp(1)`
5. `log(2.718282)`
6. `log10(100)`

5 Object Basics

An object is an item that is stored in the memory of R (*i.e.*, the current workspace).

There are many different types of objects. Here we introduce the objects most commonly used for statistical analysis and modeling in R including variables, vectors, matrices, data frames, and lists. A new object is created in R by creating a name and assigning something to it, usually using the "<-" symbol combination (the assignment arrow). For example:

```
> bob <- 20
> bob
```

```
[1] 20
```

We can name the objects anything we want, but often these variables will represent real things we want to quantify. For instance, we may want to store the infectious period of a particular pathogen. Then we have:

```
> infectious.period <- 20  # infectious period in units of days
> infectious.period
```

R code	operation
`str(x)`	return information about the structure of x
`ls()`	list all objects and functions in the workspace
`rm(x)`	remove x from the workspace
`x <- y`	assigns the value y to object x
`y -> x`	assigns the value y to object x (not recommended, the preceding form is the standard convention)
`x = y`	assigns the value y to object x (not recommended because of the potential for confusion with ==)

Table 4: Common object commands in R .

```
[1] 20
```

Note that if a number is of type "numeric" (*i.e.*, it is a variable, vector, or matrix consisting entirely of numbers) all of the arithmetic operators and built in mathematical functions introduced above can be used with it.

```
> log10(bob)
```

```
[1] 1.30103
```

```
> bob*2
```

```
[1] 40
```

Reusing "<-" on an existing object can change what is stored in the object:

```
> bob <- 40
> bob
```

```
[1] 40
```

Objects can also be used to store the results of a calculation or the output of a function:

```
> bob <- bob+2
> bob
```

```
[1] 42
```

There are a few simple commands that apply to working with objects in general. For example the "structure" command, `structure(x)` or `str(x)`, tells you what kind of object `x` is and some simple information about the elements it contains.

```
> str(bob)
```

```
 num 42
```

Exercise 10. Create a new object named "sue" that is the natural logarithm of 8.

6 Vectors

Vectors are objects that contain a string of values of the same type. Here we will focus primarily on creating and manipulating numeric vectors. However, we will also briefly consider text vectors and how they can be used to assign names in the final part of this section.

How to create a numeric vector

There are many ways to create numeric vectors in R. The simplest way to create one is using the combine function `c`:

```
> x <- c(1,3,5,7,2,8,4,4,10)
> x

[1]  1  3  5  7  2  8  4  4 10
```

Vectors consisting of sequences of numbers are easy to create because R interprets any two integers separated by a colon as the full sequence of integers that includes them:

```
> 1:10

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

> x <- c(1:10)
> x

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

Vectors consisting of a sequence of the same number or repeats of the same element can be produced using the "replicate" command `rep`:

```
> rep(2, 5)

[1] 2 2 2 2 2

> x <- rep(2,10)
> x

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

Vectors of normally distributed variables can be generated using rnorm:

```
> x <- rnorm(10)
> x
```

```
 [1] -0.86041639 -0.11268696 -1.14965218  0.51863185 -1.38396799 -0.09381101
 [7] -1.18565478  0.12052393 -0.96392911 -0.61787881
```

```
> x <- rnorm(10, mean = 5)
> x
```

```
 [1] 6.896562 4.715646 5.735207 4.700469 5.566362 3.854812 4.542656 4.970149
 [9] 3.800455 5.149200
```

A vector with elements that simulate random draws from a binomial distribution can be created using the rbinom command. When using this command three values must be specified. The first value is the number of observations (*i.e.*, the number of elements in the vector to be created), the second number is the number of trials per observation (*i.e.*, the maximum number of possible successes or failures), and the last number is the probability of success per trial (*i.e.*, the odds of getting a 1 rather than a zero).

```
> x <- rbinom(10, 1, 0.5)
> x
```

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

```
> x <- rbinom(10, 4, 0.5)
> x
```

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

Several of these methods can be combined to create complex vectors:

```
> x <- c(1,3,5,7, 1:8, rep(2,3))
> x
```

```
 [1] 1 3 5 7 1 2 3 4 5 6 7 8 2 2 2
```

Finally, you can add elements to an existing vector using the append command:

```
> x <- c(1:10)
> x
```

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

```
> x <- append(x, 11)
> x
```

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

```
> x <- append(x, x)
> x
```

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

Exercise 11. Create a vector consisting of 12 random observations of 6 trial draws from a binomial distribution where the chance of success per trial is 30%.

Exercise 12. Create a vector that consists of the number sequence 1 2 3 4 5 repeated five times. Hint: this can be done using 13 characters not including spaces.

How to select elements of a vector

There are several ways of calling elements of a vector. Brackets can be used to reference elements in a vector by position:

```
> x <- c(1:20)
> x[4]

[1] 4

> x[4:15]

 [1]  4  5  6  7  8  9 10 11 12 13 14 15
```

Brackets can also be used to remove elements from a vector, or to reference all elements of a vector apart from the ones specified:

```
> x[-4:-10]

 [1]  1  2  3 11 12 13 14 15 16 17 18 19 20

> x <- x[-4]
> x

 [1]  1  2  3  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Positions that are out of sequence can be referenced by creating a vector of positions to be called:

```
> x <- c(1:20)
> y <- c(3, 5, 12)
> x[y]

[1]  3  5 12

> x[-y]

 [1]  1  2  4  6  7  8  9 10 11 13 14 15 16 17 18 19 20
```

Finally, logical operators can be used to indicate elements of a vector that fall into some range:

```
> x <- c(1:5,2,2,2,2,2)
> x[x > 4]

[1] 5

> x[x < 4]

[1] 1 2 3 2 2 2 2 2

> which(x < 4)

[1]  1  2  3  6  7  8  9 10

> which(x == 2)

[1]  2  6  7  8  9 10
```

Exercise 13. Using the vector that you generated in Exercise 12, identify the position of all elements that are not equal to three.

Exercise 14. Using the vector that you created in Exercise 11, create a new vector where all observations lacking any successful trials are removed.

Vector Mathematics

When a function or arithmetic operation is applied to a vector, it is applied to each element of the vector:

```
>  x <- c(1:10)
>  x/2

 [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

>  x-2

 [1] -1  0  1  2  3  4  5  6  7  8

>  log10(x)

 [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700 0.7781513 0.8450980
 [8] 0.9030900 0.9542425 1.0000000

>
```

When vectors are used as variables, whatever operation performed between them is applied among corresponding positions of each vector:

```
>  x <- c(1:10)
>  y <- c(1:5,2,2,2,2,2)
>  x+y

 [1]  2  4  6  8 10  8  9 10 11 12

>  x/y

 [1] 1.0 1.0 1.0 1.0 1.0 3.0 3.5 4.0 4.5 5.0

>
```

Exercise 15. Create two vectors, `x <- rep(5, 5)` and `y <- c(1:3, 1, 2)`. Predict the result of the following operations and use R to see if you are correct.

1. `y*3`
2. `x+y`
3. `x*y`

Vector Functions

There are a number of specialized functions that are primarily used when dealing with vectors. Note, however, that many of these functions can be used with other types of objects (particularly matrices). A few examples in action:

```
>  x <- c(1,3,5,7,20,2,8,4,4,10,2,2,1,1,15,3)
>  head(x)

[1]  1  3  5  7 20  2

>  max(x)

[1] 20

>  unique(x)

 [1]  1  3  5  7 20  2  8  4 10 15

>  sort(x)

 [1]  1  1  1  2  2  2  3  3  4  4  5  7  8 10 15 20

>  mean(x)

[1] 5.5
```

R code	operation
`head(x)`	return first 6 elements x
`tail(x)`	return last 6 elements in x
`max(x)`	maximum of x
`min(x)`	minimum of x
`length(x)`	number of elements in x
`unique(x)`	number of unique values among the elements of x
`sort(x)`	x sorted from lowest to highest values
`range(x)`	minimum and maximum of x
`mean(x)`	arithmetic mean of x, equal to `sum(x)/length(x)`
`median(x)`	median of x
`sample(x)`	returns x in a random order
`sample(x, replace=TRUE)`	samples the vector x "with replacement"

Table 5: Common vector functions in R .

It is often useful to combine vector functions. For example, this combination returns the number of unique elements in vector `x` from above:

```
> length(unique(x))

[1] 10
```

A function that is particularly important for setting up randomizations is the sample function. It can be used to sample from a distribution with or without replacement. Notice that in the first example, no numbers are duplicated (*i.e.*, every element of `x` has been sampled, and no elements have been lost or "replaced") but in the second example some numbers are duplicated (*i.e.*, `x` has been sampled with replacement).

```
>  x <- c(1:20)
>  sample(x)

 [1]  4 20 12 11 14  6  5 15 13  2  9 17 19  7 10 16  3 18  8  1

>  sample(x, replace = T)

 [1]  8 12  7  8  5  6  4 14 10 11  6  7 14 11 14  4 12 20  1 11
```

Exercise 16. Create a vector `x`, `x <- c(1:10)`. Predict the result of the following operations and use R to see if you are correct.

1. `tail(x)`
2. `median(x)`
3. `length(x)`
4. `sum(x)`
5. `sum(unique(x))`

Text Vectors and Assigning Names to Vectors

Vectors can also be constructed of words. These can be used to store information such as names of species or individuals in a data frame, which we will get to later. Text vectors can also be used to assign names to the elements of a vector using the names command. The elements of a text vector need to be surrounded by quotes, or R will misinterpret them as the names of objects in the workspace. For instance, we might want a vector containing the names of some genera of bacteria.

```
> y <- c("Escherichia","Chlamydia","Bacillus","Brachyspira","Leptonema")
> y

[1] "Escherichia" "Chlamydia"   "Bacillus"    "Brachyspira" "Leptonema"
```

Now `y` can be used to assign names to a new vector `x`.

```
> x <- c(1:5)
> names(x) <- y
> x

Escherichia   Chlamydia    Bacillus Brachyspira   Leptonema
          1           2           3           4           5
```

When assigning names, it is not necessary to save the names in a vector as an intermediate step.

```
> names(x) <- c("Escherichia","Chlamydia","Bacillus","Brachyspira","Leptonema")
> x

Escherichia   Chlamydia    Bacillus Brachyspira   Leptonema
          1           2           3           4           5
```

Exercise 17. Create a vector with names corresponding to six people you know and values consisting of their total number of pets and children.

7 Matrices

A matrix is an object with rows and columns of values of the same type. Here we will focus entirely on numeric matrices.

Creating a Matrix from a Vector

Matrices can be created from vectors using the matrix command. By default the values are read in columnwise, that is values are assigned to the first column, then the second, then the third, ect.

```
> x <- c(1:25)
> y <- matrix(x, nrow=5)
> y
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

The switch `byrow` can be activated to read data in by rows instead:

```
> z <- matrix(x, byrow=T, nrow=5)
> z
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
```

Exercise 18. Create a matrix with four columns and five rows of data drawn randomly from a normal distribution.

Referencing Positions in a Matrix

Positions in a matrix are referenced using `[row, column]`.

```
> z[2,5]
```

```
[1] 10
```

If one value is omitted, an entire row or column is referenced.

```
> z[2,]
```

```
[1]  6  7  8  9 10
```

```
> z[,5]
```

```
[1]  5 10 15 20 25
```

In addition, most of the ways of referencing multiple positions in a vector can be used with matrices.

Exercise 19. Display a list of all elements in the first column of the matrix you created in Exercise 18.

Exercise 20. Using the matrix you created in Exercise 18, use a logical operator to return a list of all values in the matrix greater than zero. Is the information returned by rows or columns?

Matrix Math and Functions

The great majority of arithmetic operations give similar results with matrices to those you get from vectors.

```
> z
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
```

```
> z*2
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    4    6    8   10
[2,]   12   14   16   18   20
[3,]   22   24   26   28   30
[4,]   32   34   36   38   40
[5,]   42   44   46   48   50
```

```
> z+2
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    4    5    6    7
[2,]    8    9   10   11   12
[3,]   13   14   15   16   17
[4,]   18   19   20   21   22
[5,]   23   24   25   26   27
```

```
> z*z
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    9   16   25
[2,]   36   49   64   81  100
[3,]  121  144  169  196  225
[4,]  256  289  324  361  400
[5,]  441  484  529  576  625
```

R can also perform matrix multiplication and matrix division.

```
> z%*%z
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]  215  230  245  260  275
[2,]  490  530  570  610  650
[3,]  765  830  895  960 1025
[4,] 1040 1130 1220 1310 1400
[5,] 1315 1430 1545 1660 1775
```

```
> z%/%z
```

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

Many of the functions used with numeric vectors can be used with numeric matrices, or their rows or columns.

```
> max(z)
```

```
[1] 25
```

```
> mean(z)
```

```
[1] 13
```

```
> mean(z[2,])
```

```
[1] 8
```

```
> sample(z[2,], replace=T)
```

```
[1] 10  7  6  8  8
```

A few additional functions are generally only used with matrices.

```
> dim(z)
```

```
[1] 5 5
```

```
> t(z)  #transpose
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

```
> colMeans(z)
```

```
[1] 11 12 13 14 15
```

R code	operation
`dim(x)`	dimensions of x (numer of rows and columns of x)
`t(x)`	matrix transpose of x
`diag(x)`	the diagonal elements of x
`colSums(x)`	sums of the columns of x
`rowSums(x)`	sums of the rows of x
`colMeans(x)`	means of the columns of x
`rowMeans(x)`	means of the rows of x

Table 6: Common matrix functions in R .

Exercise 21. Create a 3x3 matrix as follows: x <- 1:9; y <- matrix(x, nrow = 3). Predict the results of the following operations, use R to see if you are correct.

1. `median(y)`
2. `dim(y)`
3. `rowSums(y)`
4. `diag(y)`
5. `y+y*2`

Adding Data to a Matrix

Data can be added to a matrix as either a new row with `rbind` or a new column with `cbind`.

```
> sums <- rowSums(z)
> z2 <- rbind(z, sums)
> z2
```

```
     [,1] [,2] [,3] [,4] [,5]
        1    2    3    4    5
        6    7    8    9   10
       11   12   13   14   15
       16   17   18   19   20
       21   22   23   24   25
sums   15   40   65   90  115
```

```
> z2 <- cbind(z, sums)
> z2
```

```
                    sums
[1,]  1  2  3  4  5   15
[2,]  6  7  8  9 10   40
[3,] 11 12 13 14 15   65
[4,] 16 17 18 19 20   90
[5,] 21 22 23 24 25  115
```

Adding names to a matrix is very similar to adding names to a vector, save that `colnames` or `rownames` is used to specify whether columns or rows are being named. Here we suppose we have some data on whether a particular bacterial genus has been found to infect particular genus of mammals. This is just a made up example, but it illustrates the point. Then we name the columns using bacterial genera and rows using mammal genera.

```
> data <- matrix(c(0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1), nrow=5)
> cols <- c("Escherichia","Chlamydia","Bacillus","Brachyspira","Leptonema")
> rows <-c("Ovis", "Canis", "Felis", "Capra", "Sus")
> colnames(data) <- cols
> rownames(data) <- rows
> data
```

```
      Escherichia Chlamydia Bacillus Brachyspira Leptonema
Ovis            0         0        0           0         0
Canis           0         1        0           1         0
Felis           0         1        1           0         0
Capra           1         1        0           0         0
Sus             0         0        0           0         1
```

Exercise 22. Using the matrix that you created in Exercise 21, add a new row of data your original matrix y containing the means of each column. Assign the following names to the rows in the new matrix: `r1, r2, r3, means`.

8 Dataframes

A matrix contains entries that are all of the same type (for example, numbers or text strings). A data frame is a data set that can combine numerous types of data (e.g., columns of text can be combined with numeric data). Normally, each column in a dataframe represents a variable and each row represents an entity for which data has been collected, for example different species of animals or individual survey respondents. Data frames can easily be imported using the `read.csv` command, which you will learn about in the next workshop. They can be created from existing vectors of data using the `data.frame` command. Missing data are indicated by `NA`.

```
> #first create some data vectors
> disease <- c("Cpox","ADisentary","Malaria","Flu","Plague")
> agent <- c("Viral","Protozoan","Protozoan","Viral","Bacterial")
> trans <- c(1,0,0,1,0)
> vector <- c(0,0,1,0,1)
> vir <- c(4,4,3,2,5)
> #now combine them into a data frame
> diseases <- data.frame(name=disease,type=agent,tmode=trans,vecborne=vector,virulence=vir)
> diseases
```

```
        name      type tmode vecborne virulence
1       Cpox     Viral     1        0         4
2 ADisentary Protozoan     0        0         4
3    Malaria Protozoan     0        1         3
4        Flu     Viral     1        0         2
5     Plague Bacterial     0        1         5
```

The `str` command is especially useful with data frames, it tells you how many observations and the names and types of variables a data frame contains.

```
> str(diseases)
```

```
'data.frame':        5 obs. of  5 variables:
 $ name     : Factor w/ 5 levels "ADisentary","Cpox",..: 2 1 4 3 5
 $ type     : Factor w/ 3 levels "Bacterial","Protozoan",..: 3 2 2 3 1
 $ tmode    : num  1 0 0 1 0
 $ vecborne : num  0 0 1 0 1
 $ virulence: num  4 4 3 2 5
```

Variables can be referenced as numerical columns the same way as matrices, or can be referenced using their variable names with `$`.

```
> diseases[,2]
```

```
[1] Viral     Protozoan Protozoan Viral     Bacterial
Levels: Bacterial Protozoan Viral
```

```
> diseases$col
```

```
NULL
```

Exercise 23. Create a data frame with observations of three variables for six cases (i.e., people): the number of pets owned, the type of pet, and the food that the pet eats. These can be real data of the pets of people you know or completely made up. Assign row names to the data frame corresponding to the first names of six individuals.

9 Lists

Lists can combine objects of any sort into one object. Lists can be used to represent the properties of an object that R is not designed to directly represent (e.g., an object of type phylo is a list of vectors and matrices that describe a phylogenetic tree). Lists are also commonly used to organize the output of a function. Objects can be combined into lists with the `list` command.

```
> stuff <- list(1:10, z2, diseases)
> str(stuff)
```

```
List of 3
 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ : num [1:5, 1:6] 1 6 11 16 21 2 7 12 17 22 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:6] "" "" "" "" ...
 $ :'data.frame':        5 obs. of  5 variables:
  ..$ name     : Factor w/ 5 levels "ADisentary","Cpox",..: 2 1 4 3 5
  ..$ type     : Factor w/ 3 levels "Bacterial","Protozoan",..: 3 2 2 3 1
```

```
  ..$ tmode    : num [1:5] 1 0 0 1 0
  ..$ vecborne : num [1:5] 0 0 1 0 1
  ..$ virulence: num [1:5] 4 4 3 2 5
```

Items of a list can be referenced using double brackets:

```
> x <- stuff[[1]]
> x

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

If names are assigned to object in a list when it is created, items in a list can also be referenced using `listname$itemname`:

```
> stuff <- list(numbers=1:10, matrix=z2, dis=diseases)
> stuff[[3]]

        name      type tmode vecborne virulence
1       Cpox     Viral     1        0         4
2 ADisentary Protozoan     0        0         4
3    Malaria Protozoan     0        1         3
4        Flu     Viral     1        0         2
5     Plague Bacterial     0        1         5

> stuff$dis

        name      type tmode vecborne virulence
1       Cpox     Viral     1        0         4
2 ADisentary Protozoan     0        0         4
3    Malaria Protozoan     0        1         3
4        Flu     Viral     1        0         2
5     Plague Bacterial     0        1         5
```

Exercise 24. Create list containing any three objects currently in your workspace, assigning names to each element of the list.

10 Writing Functions

A function is an object that carries out some operation on arguments that are provided as input. Arguments can be other objects, input data, or commands (e.g., commanding a function to turn some option on or off, or telling a statistical function which of the available models to use). In R it is relatively easy to create custom functions. Custom functions can be used to to automate tasks you perform frequently, to automate complex tasks that you perform regularly but infrequently, or to automate techniques that you develop for other users.

Functions can be created using the `function` command. The basic syntax for writing a function is:

functionname <- function(arguments) body

For example:

```
> Rounder <- function(x){
+    #this function uses the floor function to
+    #do what the round function does
+    x <- x+0.5
+    floor(x)
+  }
> ls()
```

```
 [1] "agent"             "bob"               "cols"
 [4] "data"              "disease"           "diseases"
 [7] "infectious.period" "Rounder"           "rows"
[10] "stuff"             "sums"              "trans"
[13] "vector"            "vir"               "x"
[16] "y"                 "z"                 "z2"
```

Notice that the function `Rounder` is now in the workspace. Once loaded, a custom function can be used to perform operations on numerous arguments:

```
> Rounder(3.1)
```

```
[1] 3
```

```
> Rounder(3.7)
```

```
[1] 4
```

```
> Rounder(z2*3.8)
```

```
                   sums
[1,]  4  8 11 15 19   57
[2,] 23 27 30 34 38  152
[3,] 42 46 49 53 57  247
[4,] 61 65 68 72 76  342
[5,] 80 84 87 91 95  437
```

Unfortunately, Rounder is missing a lot of the features of a really well written function. For example, when you try to use it on the wrong kind of object, you get a default error message from R that is probably not going to be very informative to a novice user.

Exercise 25. Try to apply the function `Rounder` to the `diseases` data base. What happens?

The documentation to the function also contains no information on what arguments the function can take, or what exactly the function produces as output. All of these problems can be addressed by following the Google R style guide for writing functions:

```
> #The same function as above written following Google style guidelines:
>
> Rounder2 <- function(x){
+    #this function uses the floor function to
```

```
+    #do what the round function does
+    #
+    #Args:
+    #    x: x must be of type numeric
+    #
+    #Returns:
+    #   x or all elements of x rounded to the nearest integer
+    #
+    #Error handling
+    if (is.numeric(x)==FALSE){
+      stop("x is not of type numeric")
+    }
+    x <- x+0.5
+    y <- floor(x)
+    return(y)
+  }
```

Rounder two is much more fully documented. It also returns an error message that is potentially more informative than what R produces by default when we try to use it with `diseases`.

Exercise 26. Use Rounder2 with the diseases dataframe. How has the error message changed?

The Google R style guide recommends that the title of a function start with caps and not contain a "." or "_". The body of a function starts with open curly braces at end of first line, followed by description of function, the inputs it takes (i.e., the *arguments*), and the output it generates (i.e., what it *returns*). Use the stop command to handle and describe errors. The body of a function should end with a closed curly brace on a separate line.

See the Google R style guide for more information: http://google-styleguide.googlecode.com/svn/trunk/google-r-style.htmlfunctiondocumentation

Tip: if your function requires functions from a library to work, you can call libraries from within the function using the require command:

```
> # A simple function that needs functions from the ape library to run
> # ape is a library of functions for analysis of evolutionary relationships:
>
> RandTreePlotter <- function(x){
+    #this function creates a plot of a random evolutionary tree
+    #of size x
+    require(ape)
+    tree <- rcoal(x)
+    plot(tree)
+  }
> RandTreePlotter(20)
```

Exercise 27. Create a function that will return the square root of the number of values found in a vector.

11 Flow Control

R has many of the control structures available that you would expect if you are familiar with other programming languages. They are frequently used within functions, but will also function perfectly well

embedded in a script as long as they have something to evaluate. In the notation below, expression can refer to a single line of code or a series of commands enclosed by curly braces `{}`.

`If` statements

Here we learn to use code constructions of the following form:

```
if (condition) expression
```

The way this works is that if the condition is true, the expression is executed. Otherwise the if statement will be ignored. For example:

```
> x <- 8
> if (x < 9) print("x is less than nine")

[1] "x is less than nine"

> x <- 10
> if (x < 9) print("x is less than nine")
```

`If-else` statements

Here we learn to use code constructions of a slightly different form:

`if (condition) expression1 else expression2` if the first condition is true, expression1 will be executed. Otherwise, expression2 will be executed.

```
> Isitaten <- function(x){
+  if (x == 10)
+  print("It's a ten!")
+  else print("Not a ten")
+ }
> Isitaten(11)

[1] "Not a ten"

> Isitaten(10)

[1] "It's a ten!"
```

`for` loops

Here we learn to use code constructions of the following form:

```
for (variable in set) expression
```

Execute the expression for each instance of the variable, where the variable is equal to each value within the set once. Examples:

```
> for (i in 10:20) print(2*i)

[1] 20
[1] 22
[1] 24
[1] 26
[1] 28
[1] 30
[1] 32
[1] 34
[1] 36
[1] 38
[1] 40

> y <- c(2.3, 4.4, 3.7, 1.2)
> for (i in y) print(2*i)

[1] 4.6
[1] 8.8
[1] 7.4
[1] 2.4
```

`while` loops

Here we learn to use code constructions of a slightly different form:

```
while (condition) expression
```

In this case, *while* the condition is true, repeatedly execute the expression. Once the condition is no longer true, leave the loop and move on to the next line of code if there is one.

```
> y <- 1
> while(y < 10){
+  print(y)
+  y <- y+1
+ }

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

> print("Thank goodness that's done")

[1] "Thank goodness that's done"
```

Exercise 28. Create a function that will print every integer between 1 and the number entered, but only if the number is less than 20.