Chapter 2
Overview of the R Programming Language

In this chapter we present a brief overview of the R with the goal of helping readers who are not very familiar with this language to get started.

We start with how to install R and use the development tools. We then look at the elements of the R programming language such as operators and data types. We also look at the syntax of different structures such as conditional statements and loops, along with functions. The R programming language has its share of peculiarities; we highlight some of them below. We look at installing and loading R packages from Comprehensive R Archive Network (CRAN). Finally, we will discuss the running R code outside the R console using Rscript.

2.1 Installing R

R is available on most computing platforms such as Windows, GNU/Linux, Mac OS X, and most other variants of UNIX. There are two ways of installing R: downloading an R binary and compiling R from source. The compiled binary executables for major platforms are available for download from the CRAN website http://cran.rstudio.com/.

R can also be installed using package managers. Ubuntu users can install R using the apt-get package manager.

```
$ sudo apt-get install r-base r-base-dev
```

Similarly, on Mac OS X we can install R using ports.

```
$ sudo port install R
```

Being an open source software, the source code for R is also available for download at http://cran.rstudio.com/. Advanced users can create their own binary executables by directly compiling from source as well.

There is an active community of R developers which regularly releases a new version of R. Each version also has a name assigned to it. Except for major releases, the versions of R are usually backward compatible in terms of their functionality. In this book, we use R version 3.1.0.

M. A. Pathak, *Beginning Data Science with R,*
DOI 10.1007/978-3-319-12066-9_2

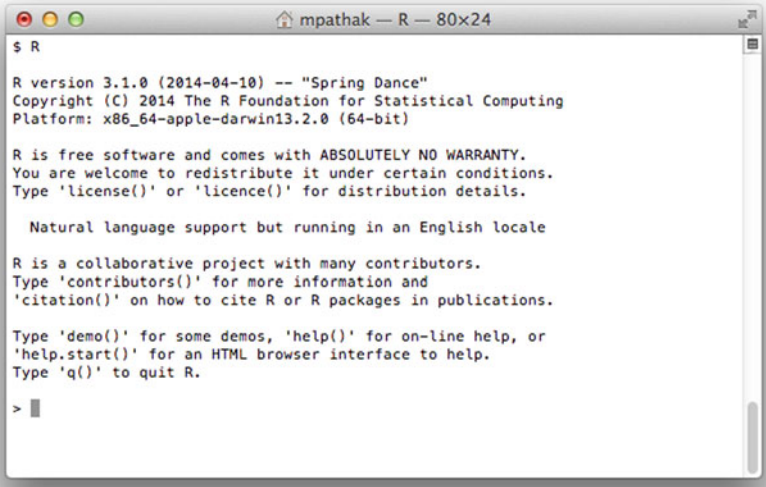

Fig. 2.1 R command-line application

2.1.1 Development Tools

The default installation of R on UNIX-based operating systems is in the form of a command-line application called `R`. We can start this application from a terminal to get a prompt as shown in Fig. 2.1. We can start typing R code at this prompt and see the output in the terminal itself. Alternatively, we can write the R code in an external text editor, and import or *source* the code in the command-line application. Many text editors such as Emacs and vi have R plugins that provide syntax highlighting and other tools. Although simplistic, this is the primary development environment for many R developers.

On Windows, R is installed as a graphical user interface (GUI) application (Fig. 2.2) with a set of development tools, such as a built-in editor. A similar GUI application called R.app also exists for Mac OS X. However, these applications are fairly basic when compared to integrated development environments (IDEs) for other programming languages.

Recently, there are many independent IDEs available for R. One of the most powerful IDEs is RStudio[1]. It is available in two editions: a GUI desktop application that runs R on the local machine, and server, where R runs on a remote server and we can interact with it via a web application.

[1] http://www.rstudio.com/.

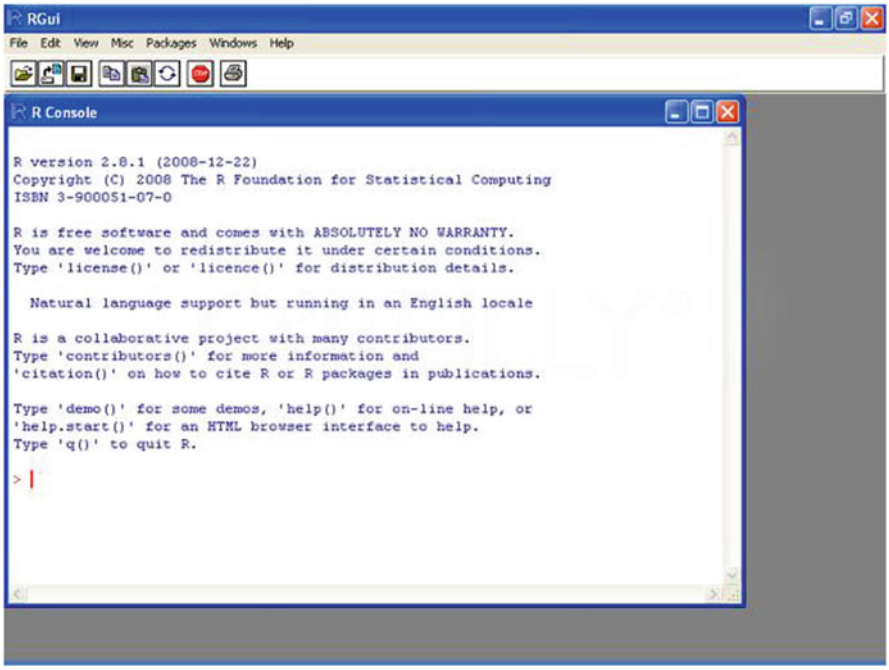

Fig. 2.2 R on Windows

2.2 R Programming Language

In this section, we briefly overview the R programming language. R is an interpreted language; the expression specified in an R program are executed line by line similar to other interpreted languages such as python or ruby rather than compiling the source code to an executable, as in C++. R is dynamically typed, which means that R infers the data types of the variables based on the context. We do not need to declare variables separately.

R has a bit of esoteric syntax as compared to most other programming languages. We look at the basic features of R below.

2.2.1 Operators

R provides arithmetic operators such as addition (+), subtraction (−), multiplication (*), division (/), and exponentiation (^). We enter the expression `3 + 4` at the R console. We do not need to terminate the expressions or lines by a semicolon.

```
> 3 + 4
[1] 7
```

As expected, R returns the answer as 7.

We use an assignment operator to assign the value of an expression to a variable. R has two assignment operators: the conventional assignment operator = which is present in most programming languages, and arrows <- and -> which are specific to R. The expression `x = 5` assigns the value 5 to x; the expression `x <- 5` and `5 -> x` have exactly the same effect. Historically, the arrow operator has been used for assignment in R. However, we do not have to use the arrow operator; apart from nonfamiliarity and typing twice as many characters, <- can also be a cause for errors: `x <- 5` vs. `x < -5`. Throughout this book, we only use the conventional assignment operator (=).

We can create expressions using variables. For instance, we assign the value 5 to the variable x and evaluate the square of x using the exponentiation (^) operator.

```
> x = 5
> x^2
[1] 25
```

R has peculiar syntax when it comes to variable names. The dot character (.) has a completely different meaning as compared to other programming languages. In R, we can use (.) in the variable names, so `x.1` and `x.2` are perfectly valid. In practice, the dot operator is used as a visual separator in variable names, similar to underscore in most other programming languages, e.g., the variable `prev_sum` is conventionally written as `prev.sum`. Underscores can also be used in R variable names as well, although they are less frequently used in practice as compared to dot.

2.2.2 Printing Values

Entering an expression on the R console evaluates the expression and then prints its value. Internally, R is calling the `print()` function with the value of the expression. We can call `print()` explicitly as well. This is useful when we want to print values for variables in a script.

```
> print(3 + 4)
[1] 7
```

Note that in the example in the previous subsection, the line `x = 5` did not print anything. R does not print the values of expressions when using the assignment operator.

The `print()` function prints the value of the expression and a new line. However, it is not straightforward to print multiple values using print(), for instance if we want to print the name of the variable along with the value. We need to print the output of the paste() function that concatenates two strings with a space.

```
> print(paste('the sum is', 3 + 4))
[1] "the sum is 7"
```

There is a simpler function called `cat()` that can print a list of values, so we do not require to call `paste()`. As the `cat()` function does not print the newline character, we need to specify it manually.

```
> cat('the sum is', 3 + 4, '\n')
the sum is 7
```

2.2.3 Basic Data Types

There are two kinds of data types: scalars that represent single-valued data or composite that represent collections of scalar data. Here we look at the scalar data types in R; we will discuss about the composite data types such as vectors and data frames in Chap. 3.

R provides multiple scalar data type formats such as numeric, integer, character, logical, and complex. The numeric data type is used to represent floating point numbers while integer data for representing only integer values. We can convert variables from numeric to integer using the `as.integer()` function.

```
> as.integer(2.56)
[1] 2
```

By default, R uses the numeric data type for integer values as well. We identify the data type of a variable using the `class()` function.

```
> x = 5
> class(x)
[1] "numeric"
```

We can also check if a variable is an integer using the is.integer() function. Such functions, as.datatype() and is.datatype() exist for all the data types mentioned above.

The character data type is used to represent strings. Unlike C or Java, R does not make a distinction between the single character char data type and multicharacter string data type. Additionally, we can use both single and double quotes to enclose strings; in this book, we primarily use single quotes.

```
> s1 = "string"
> s1
[1] "string"
> s2 = 'also a string'
> s2
[1] "also a string"
```

We convert between character and numeric variables using the `as.character()` and `as.numeric()` functions.

```
> as.character(2.5)
[1] "2.5"
```

```
> as.numeric('2.5')
[1] 2.5
```

Similar to other programming languages, R also has standard string processing functions such as computing the length of a string, finding substrings, splitting a string based on a character. The `stringr` library also provides a more consistent and easier to use set of functions for string processing.

The logical data type represents the boolean values: true and false. R uses two constants `TRUE` and `FALSE` to represent boolean values. These values are also represented by abbreviated constants `T` and `F`. In this book, we use these abbreviated constants to represent boolean values. R provides the standard boolean operators: and (`&`), or (`|`), not (`!`) along with relational operators such as equal to (`==`), less than (`<`) and greater than (`>`) that operate on numeric variables and return boolean values.

R also provides support for representing complex variables that contain a real and imaginary component.

```
> z = 2 + 3i
```

We can directly perform operations on the complex variables.

```
> z^2
[1] -5+12i
```

We will not be using complex data types in this book.

2.2.4 Control Structures

R provides control structures such as conditional branches (if-else) and loops. The syntax for if-else is similar to most other programming languages:

```
> if (x > 0) {
+     y = 'positive'
+ } else {
+     y = 'negative or zero'
+ }
```

In this case, `y` will be assigned the string `'positive'` if `x > 0` and `'negative or zero'` otherwise.

There are many other ways to write the same statement in R. Firstly, we can use if-else to return a value.

```
> y = if (x > 0) 'positive' else 'negative or zero'
```

We can also write the same expression using the `ifelse()` function, where the first argument is the boolean condition, and the second and third arguments are the respective values for the condition being true and false.

```
> y = ifelse(x > 0, 'positive', 'negative or zero')
```

An extension of the `ifelse()` function to multiple values is the `switch()` function.

R also provides multiple looping structures as well. The simplest loop is the while loop where we specify the boolean condition along with a body of steps that that are executed each time until the condition is met. The syntax for while loop is not different from that in C. We use the while loop to compute the sum of squares from 1 to 10.

```
> total = 0
> i = 1
> while (i <= 10) {
+     total = total + i^2
+     i = i + 1
+ }
> total
[1] 385
```

There are no ++ or += operators in R.

Another useful looping construct is the repeat loop, where there is no boolean condition. The loop continues until a break condition is met; conceptually, the repeat loop is similar to `while (T)`. We compute the same sum of squares from 1 to 10 using a repeat loop.

```
> total = 0
> i = 1
> repeat {
+     total = total + i^2
+     if (i == 10) break
+     i = i + 1
+ }
> total
[1] 385
```

R also has a powerful for loop that is more similar to for loop of python or Javascript as compared to the for loop in C. In this loop, we iterate over a vector of elements. We use the `in` operator to access an element of this vector at a time. We will discuss vectors in more detail in Chap. 3; for now, we construct a vector of elements from 1 to 10 as `1:10`. We compute the same sum of squares from 1 to 10 using a for loop below.

```
> total = 0
> for (i in 1:10) {
+     total = total + i^2
+ }
> total
[1] 385
```

2.2.5 Functions

R provides strong support for creating functions. In practice, most of the interactions we have with R is through functions: either provided by the base package, or user defined functions containing application logic.

The syntax for calling a function in R is similar to most other programming languages. For instance, we use the function `sum()` to compute the sum of a vector of numbers. This function is provided by the base package.

```
> sum(1:10)
[1] 55
```

R has special syntax for defining functions. As with other programming languages, we specify the function name, parameters along with body of the statements containing a return value. The difference is that we create a function using the `function` keyword and assign it to a variable. We will later see the reason for this.

We create a function called `avg()` to compute the average value of a vector of numbers. This is an implementation of the `mean()` function of the base package. We use the `length()` function that computes the number of elements or length of a vector.

```
> avg = function(x) {
+     return(sum(x)/length(x))
+ }
> avg(1:10)
[1] 5.5
```

We do not need to specify the return value explicitly in function; the last evaluated expression is automatically considered as the return value. For one line functions, we do not need to enclose the function body in braces. We can rewrite the same function as:

```
> avg = function(x) sum(x)/length(x)
> avg(1:10)
[1] 5.5
```

In R, functions are treated as first-class objects similar to other data types like numeric, character, or vector. This is a fundamental property of *functional programming* languages. Although the functional programming paradigm always had a strong niche community, it has become mainstream with the recent widespread adoption of languages like Scala, Clojure, OCaml,

The literal name of the function, in our case `avg`, corresponds to the function object, while the function call `avg(1:10)` corresponds to a value that is returned by the function when it is evaluated with the input `1:10`.

We can also assign the function sum to another variable `my.function`. Calling `my.function()` has the same effect of calling `sum()`.

```
> my.function = sum
```

```
> my.function(1:10)
[1] 55
```

Additionally, we can also pass a function as an argument to other higher-order functions. For instance, we create a function `sum.f()` that computes the sum of a vector after the function `f()` has been applied to it. Such a higher-order function should work for any function `f()`; at the time of defining `sum.f()`, we do not know what `f()` is going to be.

```
> sum.f = function(x, f) sum(f(x))
```

We compute the sum of squares of numbers between 1 and 10 using `sum.f()` below. As `f()`, we pass the square function.

```
> sum.f(1:10, function(x) x^2)
[1] 385
```

We created the function `function(x) x^2` without assigning any name to it prior to the function call. Such functions are known as anonymous functions.

R provides strong support for functional programming. There are benefits to using this paradigm including concise and cleaner code.

2.3 Packages

A self-contained collection of R functions is called as a package. This is especially useful when providing this package to other users. An R package can also contain datasets along with the dependencies. It is straightforward to create packages for our own R functions [1]. In this section we look at installing and loading other packages.

When we start R, the base package is already loaded by default. This package contains basic functions for arithmetic, input/output operations, and other simple tasks. The real power of R lies in its package library. There are thousands of packages available in the CRAN covering a very large number of data analysis methods. We can install these packages from R using the `install.packages()` function. This function downloads the source files for a package from a CRAN mirror website, builds the package, and stores the package in a local repository. The user does not need to do much for installing a package except for choosing the mirror.

As an example, we install the `stringr` package below. The `install.packages()` function provides a list of mirrors located around the world. We choose the first option: 0-Cloud.

```
> install.packages('stringr')
Installing package into /Users/mpathak/Library/R/3.1/library
(as lib is unspecified)
--- Please select a CRAN mirror for use in this session ---
CRAN mirror

 1: 0-Cloud                      2: Argentina (La Plata)
```

```
 3: Argentina (Mendoza)          4: Australia (Canberra)
 5: Australia (Melbourne)        6: Austria
 7: Belgium                      8: Brazil (BA)
 9: Brazil (PR)                 10: Brazil (RJ)
11: Brazil (SP 1)               12: Brazil (SP 2)
13: Canada (BC)                 14: Canada (NS)
15: Canada (ON)                 16: Canada (QC 1)
17: Canada (QC 2)               18: Chile
19: China (Beijing 1)           20: China (Beijing 2)
21: China (Hefei)               22: China (Xiamen)
23: Colombia (Bogota)           24: Colombia (Cali)
25: Czech Republic              26: Denmark
27: Ecuador                     28: France (Lyon 1)
29: France (Lyon 2)             30: France (Montpellier)
31: France (Paris 1)            32: France (Paris 2)
33: France (Strasbourg)         34: Germany (Berlin)
35: Germany (Bonn)              36: Germany (Goettingen)
37: Greece                      38: Hungary
39: India                       40: Indonesia (Jakarta)
41: Indonesia (Jember)          42: Iran
43: Ireland                     44: Italy (Milano)
45: Italy (Padua)               46: Italy (Palermo)
47: Japan (Hyogo)               48: Japan (Tokyo)
49: Japan (Tsukuba)             50: Korea (Seoul 1)
51: Korea (Seoul 2)             52: Lebanon
53: Mexico (Mexico City)        54: Mexico (Texcoco)
55: Netherlands (Amsterdam)     56: Netherlands (Utrecht)
57: New Zealand                 58: Norway
59: Philippines                 60: Poland
61: Portugal                    62: Russia
63: Singapore                   64: Slovakia
65: South Africa (Cape Town)    66: South Africa(Johannesburg)
67: Spain (A Corua)             68: Spain (Madrid)
69: Sweden                      70: Switzerland
71: Taiwan (Taichung)           72: Taiwan (Taipei)
73: Thailand                    74: Turkey
75: UK (Bristol)                76: UK (Cambridge)
77: UK (London)                 78: UK (London)
79: UK (St Andrews)             80: USA (CA 1)
81: USA (CA 2)                  82: USA (IA)
83: USA (IN)                    84: USA (KS)
85: USA (MD)                    86: USA (MI)
87: USA (MO)                    88: USA (OH)
89: USA (OR)                    90: USA (PA 1)
91: USA (PA 2)                  92: USA (TN)
93: USA (TX 1)                  94: USA (WA 1)
95: USA (WA 2)                  96: Venezuela
97: Vietnam

Selection: 1
trying URL 'http://cran.rstudio.com/src/contrib/
stringr_0.6.2.tar.gz' Content type 'application/x-gzip'
length 20636 bytes (20 Kb) opened URL
```

http://www.springer.com/978-3-319-12065-2

Beginning Data Science with R

Pathak, M.A.

2014, XI, 157 p. 155 illus., 26 illus. in color., Hardcover
ISBN: 978-3-319-12065-2