R

As Data Science adoption increases more in the industry, the demand for data scientists has been increasing at an astonishing pace. Data scientists are those rare breed of ‘unicorns’ who are required to be omniscient, and, according to popular culture, a data scientist is someone who know more statistics that a programmer and more programming than a statistician. One of the most important tools in a data scientist’s toolkit is the knowledge of a general purpose programming language that enables a data scientist to perform tasks of data cleaning, data manipulation, and statistical analysis with ease. Such requirements call for the programming languages that are easy enough to learn and yet powerful enough to accomplish complex coding tasks. Two such de-facto programming languages for Data Science used in industry and academia are Python and R.

While in the previous chapter we focused on basics of Python programming language, in this chapter we will focus on second most popular programming language for Data Science – R. Similar to the chapter on Python, we do not aim to cover comprehensively all topics of R, but aim to provide enough material to get the basics of R and start working with R for your daily programming tasks. A detailed knowledge of R can be gained through an excellent collection of books and Internet resources. Although prior programming experience is helpful, this chapter does not require any prior knowledge of programming.

What is R?

Similar to Python, R is a high level and general purpose programming language that was first announced in 1992 with a development version being released in 1995. R is essentially an implementation of another programming language called S, and was developed by Ross Ihaka and Robert Gentleman (and hence the name R after the initial of both creators of the language). While R is billed as a general purpose programming language, it’s core usage is in the field of statistics and data science where R enjoys a huge audience and heavy support from scientific communities.

R is an open-source language that allows anyone to contribute to R environment by creating packages and making them available to other users. R has a fairly large scientific community and is used in a variety of settings such as financial research, algorithms development, options and derivatives pricing, financial modeling, and trading systems. R was written mostly In C, Fortran and R itself, and you would see that many of R packages are written in either of these programming languages. This also means that there is good interoperability between R and these programming languages.

Why R for Data Science?

As stated at the start of this chapter, R is one of the de-facto languages when it comes to Data Science. There are a number of reasons why R is such a popular language amongst Data Scientists. Some of those reasons are listed below:

- R is a high-level general purpose programming language that can be used for varied programming tasks such as web scraping, data gathering, data cleaning and manipulation, website development, and for statistical analysis and machine learning purposes.
- R is a language that is designed mostly for non-programmers and hence is easy to learn and implement.
- R is an open source programming language. This implies that a large community of developers contribute continually to R ecosystem.
- R is easily extensible and enjoys active contribution from thousands of developers across the world. This implies that most of the programming tasks can be handled by simply calling functions in one of these packages that developers have contributed. This reduces the need for writing hundreds of lines of code, and makes development easier and faster.
- R is an interpreted language that is platform independent. As compared to some of the other programming languages, you do not have to worry about underlying hardware on which the code is going to run. Platform independence essentially ensures that your code will run in the same manner on any platform/hardware that is supported by R.

Limits of R

While R is a great programming language meant for general purpose and scientific computing tasks, R has its own set of limitations. One such limitation is that R has a relatively steeper learning curve than other programming languages such as Python. While this means increased effort for learning the language, once you have a hang of it the development becomes very fast in R. Another major limitation of R is its inefficiency in handling large datasets. For datasets that are a few hundred MB in size, R can work smoothly but as soon as datasets size increase, or computation requires creation of intermediate datasets that can take up large memory, then performance of R begins to degrade very fast. While memory management and working with large datasets is indeed a limitation of R, this can be overcome by using commercial offerings of R. Also, for many of data science needs, it might not be needed to work with large datasets.

Chapter Plan

In this series, we plan to learn R programming language, and use the features and packages present in the language for data science related purposes. Specifically, we would be learning the language constructs, programming in R, how to use these basic constructs to perform data cleaning, processing, and data manipulation tasks, and use packages developed by scientific community to perform data analysis. In addition to working with structured (numerical) data, we will also be learning about how to work with unstructured (textual) data as R has a lot of features to deal with both the domains in an efficient manner.

We will start with discussion about the basic constructs of the language such as operators, data types, conditional statements, and functions, and later we will discuss specific packages that are relevant for data analysis and research purpose. In each section, we will discuss a topic, code snippets, and exercise related to the sessions.

Installation

There are multiple ways in which you can work with R. In addition to the basic R environment (that provides R kernel as well as a GUI based editor to write and execute code statements), most people prefer to work with an Integrated Development Environment (IDE) for R. One such free and popular environment is RStudio. In this sub-section, we will demonstrate how you can install both R and RStudio.

When you working in a team environment or if your project grows in size, it is often recommended to use an IDE. Working with an IDE greatly simplifies the task of developing, testing, deploying, and managing your project in one place. You can choose to use any IDE that suits your specifics needs.

R Installation

When R can be installed on Windows, Mac OS X, and Linux based machines. In order to install R, go to the following website:

http://cran.r-project.org/

Once at the website, select the R installation specific to your system. Most of the R installations come with a GUI based installer that makes installation easy. Follow the on-screen instructions to install R on your operating system.

Once you have installed R, an R icon would be created on the Desktop of your computer. Simply double-click the icon to launch R environment.

R Studio

RStudio is a free and open source IDE for R programming language. You can install RStudio by going to the following website:

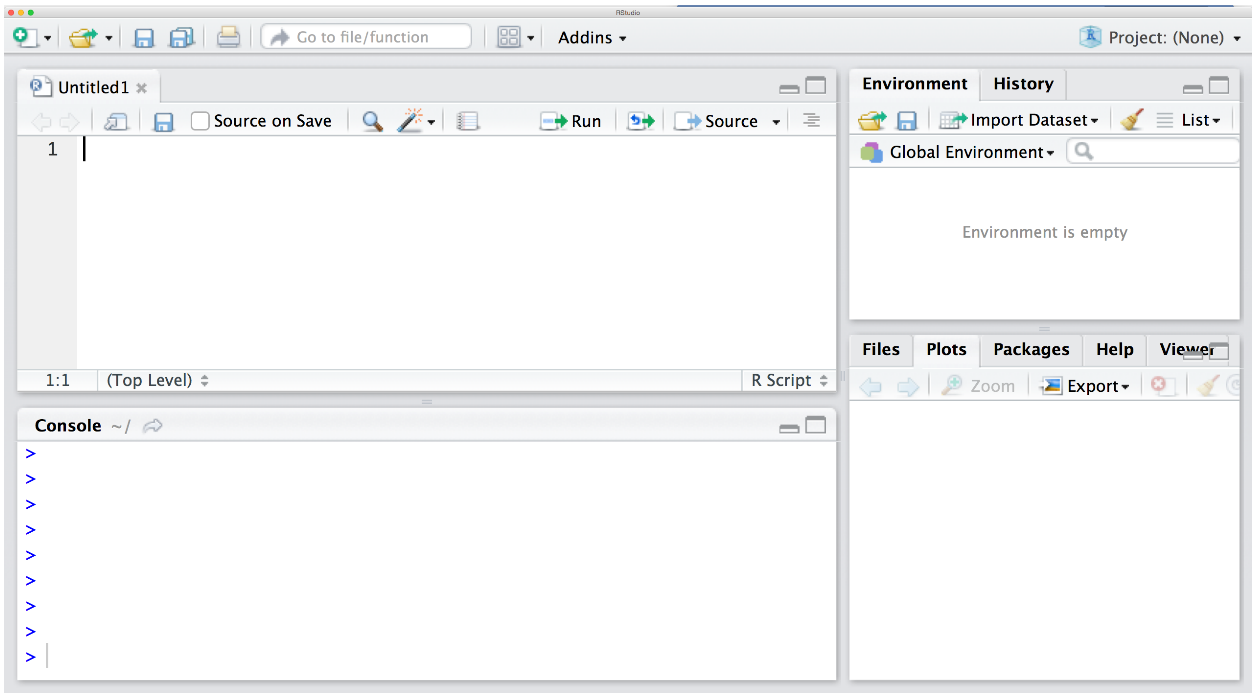
www.rstudio.com

Once at the website, download the specific installation of RStudio for your operating system. RStudio is available for Windows, Mac OSX, and Linux based systems. RStudio requires that you have installed R first so you would first need to install R before installing RStudio. Most of the RStudio installations come with a GUI based installer that makes installation easy. Follow the on-screen instructions to install RStudio on your operating system.

Once you have installed RStudio, an RStudio icon would be created on the Desktop of your computer. Simply double-click the icon to launch RStudio environment.

There are four major components in RStudio distribution:

1) A text editor at the top left hand corner. This is where you can write your R code and execute them using Run button.
2) Integrated R console at the bottom left hand corner. You can view the output of code execution in this pane and can also write individual R commands here.
3) R environment at the top right hand corner. This pane allows you to have a quick look at existing datasets and variables in your working R environment.
4) Miscellaneous pane at the bottom right hand corner. This pane has multiple tabs and provides a range of functionalities. Two of the most important tabs in this pane are Plots and Packages. Plots allow you to view the plots from code execution. In the packages tab, you can view and install R packages by simply typing the package name.



```
>
>
>
>
>
>
>
>
>
```

R Packages and CRAN

In addition to core R, most of the times you would need packages in R to get your work done. Packages are one of the most important eco-system of R. You would be using packages continuously throughout the course and in your professional lives. Good thing about R packages is that you can find most of them at a single repository: CRAN repository. In RStudio, click on Packages tab and then click on Install. A new window will open where you can start typing the name of R package that you want to install. If the package exists in CRAN repository, then you will find the corresponding name. After that simply click on Install to install the R package and its dependencies as well. This is one of the easiest ways to install and manage packages in your R distribution.

Alternatively, you can install a package from command prompt as well by using install.packages command. For example, if you type the following command, it will install e1071 package in R.

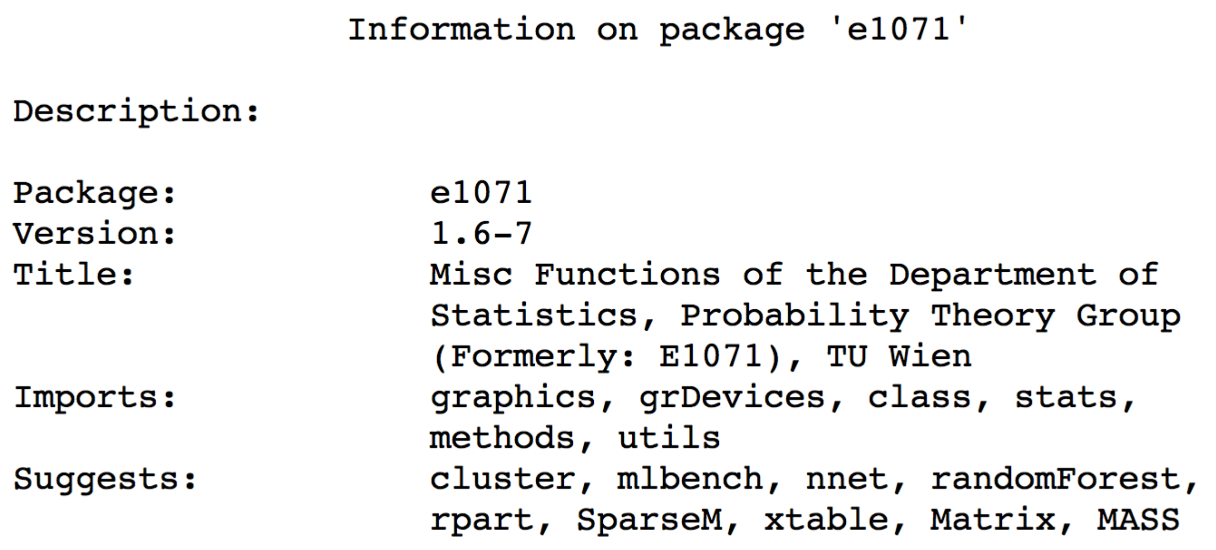
install.packages("e1071")

A not so good thing about R packages is that there is not a single place where you will get a list of all packages in R and what they do. In such cases, Internet is your best friend. Additionally, you can search for specific packages on the CRAN website. Thankfully, you will need only a handful of packages to get most of your daily work done.

In order to view contents of the package, type:

library(help=e1071)

This will give you a description about the package, as well as all available datasets and functions within that package. For example, the above command will produce the following output:



```
                 Information on package 'e1071'

Description:

Package:            e1071
Version:            1.6-7
Title:              Misc Functions of the Department of
                    Statistics, Probability Theory Group
                    (Formerly: E1071), TU Wien
Imports:            graphics, grDevices, class, stats,
                    methods, utils
Suggests:           cluster, mlbench, nnet, randomForest,
                    rpart, SparseM, xtable, Matrix, MASS
```

```
Index:

allShortestPaths        Find Shortest Paths Between All Nodes in a
                        Directed Graph
bclust                  Bagged Clustering
bincombinations         Binary Combinations
bootstrap.lca           Bootstrap Samples of LCA Results
boxplot.bclust          Boxplot of Cluster Profiles
classAgreement          Coefficients Comparing Classification
Agreement
cmeans                  Fuzzy C-Means Clustering
countpattern            Count Binary Patterns
cshell                  Fuzzy C-Shell Clustering
ddiscrete               Discrete Distribution
e1071-deprecated        Deprecated Functions in Package e1071
element                 Extract Elements of an Array
fclustIndex             Fuzzy Cluster Indexes (Validity/Performance
                        Measures)
```

Finding help in R

The simplest way to get help in R is to click on the Help button on the toolbar. Alternatively, if you know the name of the function you want help with, you just type a question mark ? at the command line prompt followed by the name of the function. For example, following commands will give you description of function solve.

```
help(solve)

?solve

?read.table
```

Sometimes you cannot remember the precise name of the function, but you know the subject on which you want help (e.g. data input in this case). Use the help.search function (without a question mark) with your query in double quotes like this

```
help.search("data input")
```

Other useful functions are find and apropos. The find function tells you what package something is in:

```
find("lowess")
```

While apropos returns a character vector giving the names of all objects in the search list that match your (potentially partial) enquiry

```
apropos("lm")
```

The R programming language

At the time of current writing, the latest version of Python available is version 3.4. Other major version of Python that is used quite extensively is version 3.2. In this chapter, we demonstrate all coding examples using version 3.2 as it is one of the most widely used versions. While there are no drastic differences in the two versions, there are some minor differences than need to be kept in mind while developing the code.

Programming in R:

Before we get started with coding in R, it is always a good idea to set your working directory in R. Working directory in R can be any normal directory on your file system and it is in this directory that all of the datasets produced will be saved. By default R sets the working directory as the directory where R was installed. You can get the current working directory by typing the following command:

```
getwd()
```

It will produce the output similar to the one below

```
[1] "/Users/rdirectory"
```

In order to change working directory, use setwd() command with the directory name as the argument:

```
setwd('/Users/anotherRDirectory')
```

This command will make the new directory you working directory.

There are two ways to write code in R: script and interactive. The scripts mode is the one that most of the programmers would be familiar with, that is all of the R code is written in one text file and the file then executes on a R interpreter. All R code files must have a .R extension. This signals the interpreter that the file contains R code. In the interactive mode, instead of writing all of the code together in one file, individual snippets of code are written in a command line shell and executed. Benefit of the interactive mode is that it gives immediate feedback for each statement, and makes program development more efficient. A typical practice is to first write snippets of code in interactive mode to test for functionality and then bundle all pieces of code in a .R file (Script mode). RStudio provides access to both modes. The top window in text editor is where you can type code in script mode and run all or some part of it. In order to run a file, just click on the Run button in the menu bar and R will execute the code contained in the file.

The bottom window in text editor acts as R interactive shell. In interactive mode what you type is immediately executed. For example, typing 1+1 will respond with 2.

Syntax Formalities

Let us now get started with understanding the syntax of R. The first thing to note about R is that it is a case sensitive language. Thus variable1 and VARIABLE1 are two different constructs in R. While we saw in the chapter on Python that indentation is one of the biggest changes that users have to grapple with, there is no such requirement of indentations in R. The code simply flows and you can either terminate the code with a semicolon or simply start writing a new code from a new line and R will understand that perfectly. We will delve more on these features as we move along in the chapter.

Calculations

Since R is designed to be a simple programming language, the easiest way to use R is as a calculator. You can simply type commands and operations in R as you would do with a calculator and R produces the output. The fundamental idea here is that one should be able to perform most of the processing tasks without worrying about the syntax of a programming language. For example, you can simply type following commands in R to get the output:

```
> log(50)

[1] 3.912023

> 5+3

[1] 8
```

Multiple expressions can be placed in single line but have to be separated by semi-colons

```
> log(20); 3*35; 5+2

[1] 2.995732
[1] 105
[1] 7

> floor(5.3)

[1] 5

> ceiling(5.3)
```

```
[1] 6
```

Comments:

Comments are required in any programming language to improve readability by humans. Comments are those sections of code that are meant for human comprehension, and are ignored by R interpreter when executing. In R, you can specify single line comments with a # sign.

1) Single line comment:

A single line comment in Python begins with a pound (#) sign. Everything after the # sign is ignored by the interpreter till then end of line.

Code:
```
print("This is code line, not a comment line")
#print("This is a comment line")
```

Output:
```
This is code line, not a comment line
```

Note that in the above code snippet, first line is the actual code that is executed whereas second line is a comment that is ignored by interpreter. A strange observation in R is that it does not have support for multi-line comments. So if you want to use multi-line comments in R then you have to individually comment each line. Fortunately, IDEs such as RStudio provide workaround for this limitation. For example, in Windows you can use CTRL+SHIFT+C to comment multiple lines of code in RStudio.

Variables:

There are some in-built data types in R for handling different kinds of data: integer, floating point, string, Boolean values, date, and time. Similar to Python, a neat feature of R is that you don’t need to mention what kind of data a variable holds; depending on the value assigned, R automatically assigns a data type to the variable.

Think of a variable as a placeholder. It is any name that can hold a value and that value can vary over time (hence the name variable). In other terms, variables are reserved locations in your machine’s memory to store different values. Whenever you specify a variable, you are actually allocating space in memory that will hold values or objects in future. These variables continue to exist till the program is running. Depending on the type of data a variable has, the interpreter will assign the required amount of memory for that variable. This implies that memory of a variable can increase or decrease dynamically depending on what type of data the variable has at the moment. You create a variable by specifying a name to the variable, and then by assigning a value to the variable by using equal sign (=) operator.

Code:

```
variable1 = 100         # Variable that holds integer value
distance   = 1500.0      # Variable that holds floating point value
institute  = "ISB"       # Variable that holds a string
```

Output:

```
100
1500.0
ISB
```

Code:

```
a = 0
b = 2
c = "0"
print(a  + b)
print(c)
```

Output:

```
2
“0”
```

Naming Conventions for a Variable:

Although a variable can be named almost anything, there are certain naming conventions that should be followed:

- Variable names in R are case-sensitive. This means that Variable and variable are two different variables.
- A variable name cannot begin with a number.
- Remainder of the variable can contain any combination of alphabets, digits, and underscore characters.
- A variable name cannot contain blank spaces

```
> x <- 5
> y = 5

> print(x)
[1] 5

> print(y)
[1] 5
```

[1] indicates that x and y are vectors and 5 is the first element of the vector

Notice the use of <- for assignment operator. Assignments in R are conventionally done using <- operator (although you can use = operator as well). For most of the cases, there is no difference between the two, however in some of the specialized cases, you can get different results based on which operator you are using. The official and correct assignment operator that is endorsed is <- operator and we would encourage the readers to use the same for their coding as well.

Basic Data Types:

In addition to complex data types, R has five atomic (basic) data types. They are Numeric, Character, Integer, Complex, and Logical respectively. Let us understand them one by one.

Numbers are used to hold numerical values. There are four types of numbers that are supported in R: integer, long integer, floating point (decimals) and complex numbers.

1) **Integer:** An integer type can hold integer values such as 1, 4, 1000, -52534 etc. In R, integers have a bit length of minimum 32 bits. This means that an integer data type can hold values in the range -2,147,483,648 to 2,147,483,647. An integer is stored internally as a string of digits. An integer can only contain digits and cannot have any characters or punctuations such as $.

   Code:
```
>>> 120+200
320
>>> 180-42
138
>>> 15*8
120
```

2) Long Integer: Simple integers have a limit on the value that they can contain. Sometimes the need arises for holding a value that is outside the range of integer numbers. In such a case, we make use of Long Integer data types. Long Integer data types do not have a limit on the length of data they can contain. A downside of such data types is that they consume more memory and are slow during computations. Use Long Integer data types only when you have the absolute need for it.

Code:

```
>>> 2**32
4294967296L
```

3) Floating-Point Numbers: Floating point data types are used to contains decimal values such as fractions.

4) Complex Numbers: Complex number data types are used to hold complex numbers. In Data Science, complex numbers are used rarely and unless you are dealing with abstract math there would be no need to use complex numbers.

Vector:

Whenever you define a variable in R that can contain one of the above mentioned atomic data types, that variable would most likely be a vector. A vector in R is a variable that can contain one of more values of the same type (Numeric, Character, Logical and so on). A vector in R is analogous to an array in C or Java with the difference that we do not have to create the array explicitly and we also do not have to worry about increasing or decreasing length of array. A primary reason behind having vectors as the basic variable in R is that most of the times peope would not be working with a single value but a bunch of values in a dataset (think of a column in a spreadsheet). Thus, in order to mimic that behaviour, R implements vector. A vector can also contain single values (in such case, it would be a vector of length one). For example, all of the below variables are vectors of length one (since they contain only one element):

```
> a <- 4
> a
[1] 4

> str <- "abc"
> str
[1] "abc"

> boolean <- TRUE
> boolean
[1] TRUE
```

If you want to combine multiple values to create a vector, then you can make use of c operator in R. c() stands for concatenate operator, it’s job is to take individual elements and create a vector by putting them together. For example:

```
> x <- c(1, 0.5, 4)
> x
[1] 1.0 0.5 4.0

> y <- c("a","b","c")
> y
[1] "a" "b" "c"

> z <- vector("numeric",length=50)
> z
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[34] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Note that in the last statement, we made use of vector() function to create a function. Vector is an inbuilt function in R that will create a vector of a specific size (specified by length argument) and type (specified by numeric). If we do not specify default values for the vector, then it will take default values for the specified vector type (for example, default value for numeric is 0).

You can perform a range of functions on the vector. For example:

```
#To find the class of a vector, use class function
> class(y)
[1] "character"

> #Length of a vector
> length(y)
[1] 3
```

This representation of data in a vector allows you to ask mathematical questions easily. For example:

```
> mean(x)

[1] 1.833333

> max(x)

[1] 4

> quantile(x)

  0%  25%  50%  75% 100%

0.50 0.75 1.00 2.50 4.00
```

Vectors are quite flexible in R and you can create them in a range of ways. One very useful operator in R for vectors is sequence operator (:). A sequence operator works like an increment operator that will start with an initial value, increment in steps (default is 1), and stop at a terminal value. In doing so, the increment operator will create a vector from initial to terminal value. For example:

```
> x <- 1:50

> x

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

[23] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

[45] 45 46 47 48 49 50
```

```
> seq(0,8,0.2)

 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0

[17] 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0 5.2 5.4 5.6 5.8 6.0 6.2

[33] 6.4 6.6 6.8 7.0 7.2 7.4 7.6 7.8 8.0
```

Note that in above command, we explicitly called the seq() function (it is similar to sequence operator). The seq() function takes the initial value and terminal value as 0 and 8 respectively, and creates a vector of values by incrementing in the steps of 0.2

If we want to generate a vector of repetitive values, then we can do so easily by using rep() function. For example:

```
> rep(4,9)

[1] 4 4 4 4 4 4 4 4 4

> rep(1:7,10)

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

[34] 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3

[67] 4 5 6 7

> rep(1:7,each=3)

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

In the first case, rep function repeated value 4 nine time. In the second command, rep repeated the sequence 1 to 7 ten times. In the third, we created a vector where each value from 1 to 7 was repeated three times.

Vector Arithmetic and Processing:

You can perform the arithematic operations on vectors in a manner similar to variable operations. Here, the operations are performed on each corresponding elements:

```
> x <- c(1, 0.5, 4)

> x

[1] 1.0 0.5 4.0
```

```
> y <- c(5,3,2)

> y

[1] 5 3 2

> x+y

[1] 6.0 3.5 6.0
```

What would happen in the following case?

```
> x

[1] 1.0 0.5 4.0

> y <- c(5,3,2,1)

> y

[1] 5 3 2 1

> x+y

[1] 6.0 3.5 6.0 2.0

Warning message:

In x + y : longer object length is not a multiple of shorter object length
```

You would expect that there should be an error since the vectors are not of same length. However, while we received a warning saying that vectors are not of same length, R nevertheless performs the operation in a manner such that when the vector of shorter length finishes then the whole process starts again from first element for the short vector. This means that x in our case is the vector with three elements. While first three elements of x are added to three elements of y, but for the fourth element of y, the element from x is the first elements (since the process repeats itself for the shorter length vector). This is a peculiar behaviour of R that one needs to be careful about. If we are not careful about length of vectors while performing arithmetic operations then the results can be erroneous and can go undetected (since R does not produce any errors).

Since a vector can be viewed as an array of individual elements, we can extract individual elements of a vector and can also access sub-vectors from a vector. The syntax for doing so is very similar to what we use in Python i.e. specify the name of the vector followed by the index in square brackets. One point to be careful about is that indexes in R start from 1 (and not from 0 as in Python). For example:

```
> a <- c(1,3,2,4,5,2,4,2,6,4,5,3)
> a
 [1] 1 3 2 4 5 2 4 2 6 4 5 3

> #Extract individual elements of a vector
> a[1]
[1] 1

> #Access multiple values of a vector
> b <- a[c(1,4)]
> b
[1] 1 4

> d <- a[1:4]
> d
[1] 1 3 2 4
```

Let us say that you want to select subset of a vector based on a condition.

```
> anyvector <- a>3
> a[anyvector]
[1] 4 5 4 6 4 5

> x <- 1:30
> x[x>5]
 [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
[23] 28 29 30
```

You can also apply set theory operations (in addition to usual arithmetic operators) on vectors

```
> setA <- c("a", "b", "c", "d", "e")
> setB <- c("d", "e", "f", "g")

> union(setA, setB)
[1] "a" "b" "c" "d" "e" "f" "g"

> intersect(setA, setB)
[1] "d" "e"

> setdiff(setA,setB)
[1] "a" "b" "c"
```

Lists:

While vectors in R are a convenient way of playing with a number of values at the same time, often times the need would arise that we need to have values of different type in a vector. For example, we might want to have numeric as well as character values in the same variable. Since we cannot do so with vectors, the data type that comes to our rescue is list. A list in R is nothing but a special type of vector that can contain different types of data. We define list with a list() function in R.

```
> x <- list(1,"c",FALSE)

> x

[[1]]

[1] 1

[[2]]

[1] "c"

[[3]]

[1] FALSE

> x[3]

[[1]]

[1] FALSE

> x[1:2]

[[1]]

[1] 1

[[2]]

[1] "c"
```

In the above case, we defined a list x that contains three elements – numeric 1, character c, and a logical value FALSE.

We can then access individual elements of a list in the similar way we can do so with vectors. In addition to containing basic data types, a list can contain complex datatypes as well (such as nested lists). For example:

```
> x <- list(col1=1:3, col2 = 4)

> x

$col1

[1] 1 2 3

$col2

[1] 4

> x[1]

$col1

[1] 1 2 3

> x[[1]]

[1] 1 2 3

> x$col1

[1] 1 2 3

> x$col1[2]

[1] 2

> x[[1]][2]

[1] 2
```

In the above example, we defined a list x that contains two elements – col1 and col2. Col1 and col2 are lists by themselves – col1 contains numbers 1,2,3; and col2 contains a single element 4. You can access individual elements of a list or elements with the elements by using square brackets and the index of elements.

Matrices:

Lists and vectors are uni-dimensional objects i.e. a vector can contain a number of values and we can think of it as a single column in a spreadsheet. But if we need to have multiple columns, then vectors are not a convenient way of working around. For this R provides two different data structures at our disposal- matrices and dataframes. We will first talk about matrices and then move to dataframes.

A matrix in R is nothing but a multi-dimensional object where each dimension is an array. There are multiple ways of creating a matrix in R:

```
> m1 <- matrix(nrow=4, ncol=5)

> m1

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

[1,]   NA   NA   NA   NA   NA

[2,]   NA   NA   NA   NA   NA

[3,]   NA   NA   NA   NA   NA

[4,]   NA   NA   NA   NA   NA

> dim(m1)

[1] 4 5

> m1 <- matrix(1:10,nrow=2, ncol=5)

> m1

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

[1,]    1    3    5    7    9

[2,]    2    4    6    8   10

> dim(m1)

[1] 2 5

> matrix(data=c(1, 2, 3, 4), byrow=TRUE, nrow=2)

   [,1] [,2]
```

```
[1,]    1    2

[2,]    3    4
```

In the first example, we created a 4*5 matrix (where we specified number of rows by nrow and number of columns by ncol argument respectively) by calling the matrix function. Since we did not specify any values to be populated for the matrix, it had all NA values (default values). If we want to identify the dimensions of a matrix (its rows and columns) then we can make use of dim() function.

In the second example, we created a 2*5 matrix and also specified the values that were to be populated in the matrix (values from 1:10 specified by the sequence). The values would be filled column wise (i.e. first column will get values followed by second column and so on). If we wanted to fill values by row then we have to specify the argument byrow=TRUE (as we did in the third example).

Just like you can access individual elements of a vector, we can access rows, columns, and individual elements of a matrix using the similar notation for vectors. For example:

```
> x<- matrix(1:10,2,5)

> x

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

[1,]    1    3    5    7    9

[2,]    2    4    6    8   10

> x[1,1]

[1] 1

> x[1,]

[1] 1 3 5 7 9

> x[,2]

[1] 3 4
```

Often times it might happen that we have some vectors at our disposal and we want to create a matrix by combining those vectors. This can be done by making use of rbind and cbind operators. While rbind will join the columns by row, cbind will join the columns otherwise. For example:

```
> x<- 1:6

> x

[1] 1 2 3 4 5 6

> y <- 12:17

> y

[1] 12 13 14 15 16 17

> cbind(x,y)

    x  y

[1,] 1 12

[2,] 2 13

[3,] 3 14

[4,] 4 15

[5,] 5 16

[6,] 6 17

> rbind(x,y)

  [,1] [,2] [,3] [,4] [,5] [,6]

x    1    2    3    4    5    6

y   12   13   14   15   16   17
```

In addition to the usual arithmetic, matrices come in handy when we have to perform matrix arithmetic. The real use of matrices occurs in those situations where data is numeric in nature and we are dealing with a large set of numbers upon whom we want to perform matrix arithmetic computations. By design matrices have limited scope outside of numbers since they are not designed to be much useful for anything other than numeric data. If we want to exploit the true spreadsheet capability that we experience in Excel, then we need to use dataframes in R.

DataFrame:

A primary reason why Excel is very useful for us is that everything is laid out in neat tabular structure and this enables us to perform a variety of operations on the tabular data. Additionally, we can also hold string, logical and other types of data. This capability is not lost for us in R and is instead provided by dataframe in R.

Tabular data in R is read into a type of data structure known as data frame. All variables in a data frame are stored as separate columns, and is different from matrix in the sense that each column can be of a different type. Almost always, when you import data from an external data source, you import it using a data frame. A dataframe in R can be created using the function data.frame()

```
> x <- data.frame(col1=1:20, col2 = c(T, F, F, T))

> x

   col1  col2

1     1  TRUE

2     2 FALSE

3     3 FALSE

4     4  TRUE

5     5  TRUE

6     6 FALSE

7     7 FALSE

8     8  TRUE

9     9  TRUE

10   10 FALSE
```

```
11   11 FALSE

12   12  TRUE

13   13  TRUE

14   14 FALSE

15   15 FALSE

16   16  TRUE

17   17  TRUE

18   18 FALSE

19   19 FALSE

20   20  TRUE

> nrow(x)

[1] 20

> ncol(x)

[1] 2

> #Check structure of a data frame

> str(x)

'data.frame':   20 obs. of  2 variables:

 $ col1: int  1 2 3 4 5 6 7 8 9 10 ...

 $ col2: logi  TRUE FALSE FALSE TRUE TRUE FALSE ...
```

In the first code snippet, we specified that we are creating a dataframe that has two columns (col1 and col2). To find number of rows and columns in a dataframe, we use arguments nrow() and ncol() respectively. In order to check the structure of a data frame (number of observations, number and types of columns), we make use of function str().

Similar to matrices, we can select individual columns, rows and values in a dataframe. For example:

```
> x[1]
   col1
1     1
2     2
3     3
4     4
5     5
6     6
7     7
8     8
9     9
10   10
11   11
12   12
13   13
14   14
15   15
16   16
17   17
18   18
19   19
20   20
> x[1,1]
```

```
[1] 1

> x[,2]

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

[12]  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE

> x[2:5,1]

[1] 2 3 4 5
```

Additionally, we can also make use of $ operator to access specific columns of a dataframe. The syntax is dataframe$colname.

```
x$col1
```

R Operators:

Operators in R perform operations on two variables/data values. Depending on what type of data the variable contains, the operations performed by the same operator could differ. Listed below are the different operators in R:

+ (plus) : It would add two numbers or variables if they are numbers. If the variables are string, then they would be concatenated. For example:
 4 + 6 would yield 10. 'Hey' + 'Hi' would yield 'HeyHi'.
- (minus) It would subtract two variables.
*** (multiply)** It would multiply two variables if they are numbers. If the variables are strings/lists, then they would be repeated by a said number of times. For example:
 3 * 6 would yield 18. 'ab * 4 would yield 'abababab'.
**** (power)** It computed x raised to power y.
 4 ** 3 would yield 64 (i.e. 4 * 4 * 4)
/ (divide) It would divide x by y
// (floor division) It would give the floor in a division operation
 5 // 2 would yield 1.
% (modulo) Returns the remainder of the division
 8 % 3 gives 2. -25.5 % 2.25 gives 1.5.
< (less than) Returns whether x is less than y. All comparison operators return True or False. Note the capitalization of these names.
 5 < 3 gives False and 3 < 5 gives True.
> (greater than) Returns whether x is greater than y
 5 > 3 returns True. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns False.
<= (less than or equal to) Returns whether x is less than or equal to y

x = 3; y = 6; x <= y returns True.

>= (greater than or equal to) Returns whether x is greater than or equal to y

x = 4; y = 3; x >= 3 returns True.

== (equal to) Compares if the objects are equal

x = 2; y = 2; x == y returns True.

x = ’str’; y = ’stR’; x == y returns False.

x = ’str’; y = ’str’; x == y returns True.

!= (not equal to) Compares if the objects are not equal

x = 2; y = 3; x != y returns True.

not (boolean NOT) If x is True, it returns False. If x is False, it returns True.

x = True; not x returns False.

and (boolean AND) x and y returns False if x is False, else it returns evaluation of y

x = False; y = True; x and y returns False since x is False

or (boolean OR) If x is True, it returns True, else it returns evaluation of y

x = True; y = False; x or y returns True.

Conditional statements

After the discussion of variables and data types in R, let us now focus on the second building block of any programming language i.e. conditional statements. Conditional statements are branches in a code that are executed if a condition associated with the conditional statements is true. There can be many different types of conditional statement, however, the most prominent ones are if, while, and for. In the following sections, we discuss these conditional statements.

The if statement:

We use an if loop whenever there is a need to evaluate a condition once. If the condition is evaluated to be true, then the code block associated with if condition is executed, otherwise the interpreter skips the corresponding code block. The condition along with the associated set of statements are called the if loop or if block. In addition to the if condition, we can also specify an else block that is executed if the if condition is not successful. Please note that the else block is entirely optional.

Code:
```
x <- 0
if (x < 0) {
  print("Negative number")
} else if (x > 0) {
  print("Positive number")
} else
  print("Zero")
```

Ouput:
```
[1] "Zero"
```

The while loop:

Whereas an if loop allows you to evaluate the condition once, the while loop allows you to evaluate a condition multiple number of times depending on a counter or variable that keeps track of the condition being evaluated. Hence, you can executed the associated block of statements multiple times in a while block. Similar to an if loop, you can have an optional else loop in the while block.

Code:
```
a <- 10
while (a>0){
  print(a)
  a<-a-1
}
```
Output:
```
[1] 10
[1] 9
[1] 8
[1] 7
[1] 6
[1] 5
[1] 4
[1] 3
[1] 2
[1] 1
```

For loop:

In many ways for loop is similar to a while loop in the sense that it allows you to iterate the loop multiple times depending on the condition being evaluated. However, for loop is more efficient in the sense that we do not have to keep count of incrementing or decrementing the counter of condition being evaluated. In the while loop, onus is on the user to increment/decrement the counter otherwise the loop runs until infinity. However, in for loop the loop itself takes care of the increment/decrement.

Code:
```
for (j in 1:5){
  print(j)
}
```

Output:
```
[1] 1
[1] 2
```

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

In the code snippet below, we make use of seq_along() function that acts as a sequence of non-numeric values. The function will iterate through each of values in the specified vector x and the print loop will then print the values.

```
x <- c("a","c","d")
for (i in seq_along(x)){
  print(x[i])
}
```

```
[1] "a"
[1] "c"
[1] "d"
```

We can alternatively write the same code in following manner:

```
for (letter in x){
  print(letter)
}
```

```
[1] "a"
[1] "c"
[1] "d"
```

File Input and Output

Most of the times, in addition to using variables and in-built data structures, we would be working with external files to get data input and to write output to. For this purpose, R provides functions for file opening and closing. There are a range of funtions for reading data into R. read.table and read.csv are most common for tables, readLines for text data, and load for workspaces. Similarly, for write use write.table, write.lines.

read.table is the most versatile and powerful function for reading data from external sources. You can use it to read data from any type of delimited text files such as tab, comma and so on. The syntax is as follows:

```
inputdata <- read.table("inputdata.txt",header=TRUE)
```

In the above code snippet, we read in a file inputdata.txt into a dataframe inputdata. By specifying the argument header=TRUE, we are specifying that the input file contains the first line as header.

While you can import data using read.table function as well, there are specific functions for csv and excel files:

```
titanicdata <- read.csv("train.csv")
```

```
datafile1 <- read.table("train.csv",header=TRUE,sep=",")
```

Similar to the functions for reading files in R, there are functions for writing back data frames to R. Here are some of the most common examples that you would encounter. This list is not exhaustive and there are many more functions available for working with different file types.

```
write.csv(titanicdata,"D:// file1.csv")
```

In the above code snippet, we write the contents of dataframe titanicdata to an output file (file1.csv) in D drive.

Function:

While R is a great programming language with a number of in-built functions (such as those for printing, file reads and writes), often times you would need to write your own piece of functionality that is not available elsewhere (for example, you might want to write a specific piece of logic pertaining to your business). For such cases, rather than writing the same code again at multiple places in code, we make use of functions. Functions are nothing but reusable code pieces that need to be written once, and can then be called using their names elsewhere in the code. When we need to create a function, we need to give a name to the function and the associated code block that would be run whenever the function is called. A function name follows the same naming conventions as a variable name. Any function is defined using keyword function. This tell interpreter that the following piece of code is a function. After function, we write the arguments that the function would expect within the parenthesis. Following this, we would write the code block that would be executed every time the function is called.

In the code snippet below, we define a function named func1 that takes two arguments a and b. The function body does a sum of the arguments a and b that the function receives.

Code:

```
func1 <- function(a,b){
  a+b
}
```

```
func1(5,10) #call the function by calling function name and providing the arguments
```

Output:

```
[1] 15
```

```
square.it <- function(x) {
  square <- x * x
  return(square)
}

square.it(5)
[1] 25
```

In the above mentioned code snippet, we created a function ‘square.it using the syntax of the function. In this case, the function expects one argument and does a square of that argument. return() statement will pass the value computed to the calling line in the code (the line or variable that called the function). Note that the names given in the function definition are called **parameters** whereas the values you supply in the function call are called **arguments**.