CHAPTER 2

Writing a Function in R

The simplest function takes just one line. Below, a function called **dd** is generated, which doubles any input value.

```
> dd<-function(x)x*2
```

To call it, just treat it as any embedded R function, such as **min()** or **max()**.

```
> dd(2.45)
[1] 4.9
> dd(2.14)
[1] 4.28
```

The structure of a user-defined function is a function name (such as **dd** in the above case) followed by **<-function()**. We insert our input values in the parentheses. The last part will be the body of our function.

2.1. Introduction

Most of the computations carried out in R involve evaluation of various functions, supplied by R via various packages (We devote two chapters to discuss R packages—see chapters 16 and 31) or written by individual users. In this chapter, we focus on how to write simple functions. After familiar ourselves with the basic structure—how to input values, how to add comments—we will use R as a financial calculator. In other words, we could write most finance-associated functions in R according to their formulae presented in chapter 1. It is straightforward to call them. For example, calling **pv_f(100,0.1,2)**, we calculate the present value of $100 received in 2 years with a 10% annual interest rate. In the next several chapters, we will explore this further by analyzing nonstandard and more complex functions, such as the Black-Sholes- Merton option model.

2.2. Writing a Simple One-Line Function in R

Below, a present-value function is generated with just one line.

```
> pv_f<-function(fv,r,n) fv(1+r)^n
```

We could write a function without any input. For instance, type **q()** to quit R. If you type **exit()**, you will get an error message.

```
> exit()
Error: could not find function "exit"
```

Thus, we could write a function called **exit()**, which is equivalent to **q()**.

```
> exit<-function() q()
```

2.3. Input Values: Positional Arguments, Keyword Arguments, and Mixed Ones

To call a function, we have three methods to input values: positional, keyword, or mixed.

Positional Arguments

The meaning of an input variable depends on its position. What is the present value of $100 occurring in one year with a discount rate of 8%? According to the structure of the **pv_f()**, the order of inputs is a future value, an effective period rate, and the number of periods.

```
> pv_f(100,0.08,1)
[1] 92.59259
```

If we have the following order of inputs, **pv_f(1,100,0.08)**, then the first input value of the future value will be $1, the interest rate is 10,000%, and the number of periods is 0.08. The final result will be $0.69 instead of $92.59. Thus, we have to be careful with the order of our input variables.

```
> pv_f(1,100,0.08)
[1] 0.6912805
> 1/(1+100)^0.08
[1] 0.6912805
```

Keyword Approach

Specify a keyword in front of each argument such as **fv=100**.

```
> pv_f(fv=100,n=1,r=0.08)
[1] 92.5926
```

The advantage of the keyword approach is that the order of input variables no longer plays a role. See the following three equivalent ways of inputting data.

```
> pv_f(fv=100,n=2,r=0.1)
[1] 82.64463
> pv_f(n=2,fv=100,r=0.1)
[1] 82.64463
> pv_f(r=0.1,fv=100,n=2)
[1] 82.64463
```

To view all *user-defined* functions, type **ls()**.

```
> ls()
[1] "dd" "pv_f"
```

To know the structure of a specific function, simply type its name, such as **pv_f**.

```
> pv_f
function(fv,r,n)fv/(1+r)^n
```

Mixed Approach

The knowledge about the above two methods should suffice. However, for the completeness, we introduce this method, which has the following procedure.

- Rule 1. Exact match: Match keywords from your input variables that are the exactly same as the keywords specified by the function.
- Rule 2. Partial match: After rule 1, partially match keywords.
- Rule 3. Positional match: After rules 1 and 2, anything left will be matched according to the position of each input variable.

Assume that we have a function with the following form.

```
>my_f<-function(x,y,aa,aabb){
cat("x=",x, "\n")
}
```

Case 1 is given below.

```
# case 1
> my_f(aa=1,2,3,4) # according to rule 1: aa=1
# according to rule 3: x=2, y=3, aabb=4
```

Here is the case 2.

```
# case 2
> my_f(aabb=1,a=2,3,4) # according to rule 1: aabb=1
# according to rule 2: aa=2
# according to rule 3: x=3, y=4
```

We can check our results with the following codes.

```
# use this function to check
function(x,y,aa,aabb) {
cat("x=",x, "\n")
cat("y=",y,"\n")
cat("aa=",aa,"\n")
cat("aabb=",aabb,"\n")
}
```

2.4. Programs with Multiple Lines

For a multiple-line function, a pair of curly braces, { and }, is used to embrace those lines.

```
my_function<-function(x,y,z){
# line 1
# line 2
# line 3
# more
}
```

For those simplest one-line functions, we still can add a pair of curly braces to circle the main function body.

```
pv_f<-function(fv,r,n) {
fv*(1+r)^(-n)
}
```

To make our functions self-explanatory, the best strategy is to add a few comments such as the objective of the program, definitions of input variables, plus one or two examples on how to apply the function.

```
pv_perpetuity<-function(c,r){
"Objective: estimate the present value of perpetuity
c : cash flow (1st at the end of 1st period
r : effective period rate
e.g.,
> pv_perpetuity(10,0.08)
[1] 125
"
return(c/r)
}
```

In the above program, the last line before the second curly brace, we use **return()**. This is a standard command of returning our final result. Adding **return()** makes our programs clearer; even its omission causes no error. In R, there are two ways to add a comment. The number sign (#) indicates that the rest of the line will be a comment. The only exception is when # is a part of a string, such as **x<-"the # of observations is 100"**.

The R complier will ignore comments when it compiles the program. For a multipleline comment, it is cumbersome to apply many number signs since we have to add one in front of each comment line. In those cases, a pair of double quotation marks is used to encircle all those comment lines (see the above example).

2.5. Well-Indented Codes Are More Readable

The following two programs are essentially the same except indentation. Obviously the first one is easier to read.

```
pv_perpetuity_due<-function(c,r){
    "Objective: estimate the present value of a perpetuity
                c : cash flow (1st at the end of 1st period
                r : effective period discount rate
                 e.g.,
               > pv_perpetuity(10,0.08)
                   [1] 125
      "
     return(c/r*(1+r))
}
```

The second program below has the same codes.

```
pv_perpetuity_due<-function(c,r){
"Objective: estimate the present value of a perpetuity
c : cash flow (1st at the end of 1st period
r : effective period discount rate
e.g.,
 > pv_perpetuity(10,0.08)
[1] 125
"
return(c/r*(1+r))
}
```

For a simple function such as the example above, the effectiveness of a good indentation is less obvious. However, when we have a more complex program, such as multiple loops and blocks, a proper indentation will be more critical.

2.6. Using Notepad as a Text Editor

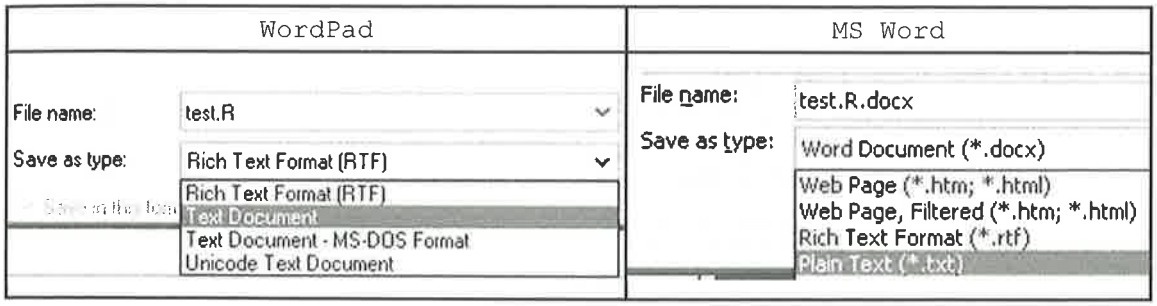
There are several ways to initiate Notepad.

1. Click **Start** then **Notepad**.
2. Click **Start** then enter *notepad*.

Another way to make our lives a bit easier is to put Notepad on our desktops. Alternatively we can use the editor included in R to generate a new program or modify our existing programs.

```
# For a new file, "File" -> "New script"
# To edit, click "File" -> "Open script"
```

In addition, we can use WordPad, MS Word, or other word editors to write or edit our programs. Just remember to apply a text format when saving our programs.



2.7. Extensions of R Programs Are Not Critical

When writing an R program, its extension is not important. For example, we could name it **test.txt**, **test.R**, or just **test** (i.e., completely ignoring the extension). The advantage of a **.txt** extension is that our computers will automatically launch Notepad to open the program when we click it. Similarly, if we intend to use the R editor to open our programs, it is a good idea to adopt a **.R** extension. Another advantage of using **.R** as an extension is it distinguishes our programs with other files such as input, output, or data files with a **.txt** extension.

2.8. How to Run an R Program

Assume that our program has the following three lines for three functions.

```
pv_f<-function(fv,r,n)fv/(1+r)^n
fv_f<-function(pv,r,n) pv*(1+r)^n
pv_perpetuity<-function(c,r)c/r
```

Method 1: Copy and Paste

First, we use Notepad to generate those three lines then highlight them and paste them to the R console. A careful reader would find that this procedure is equivalent to typing those three lines on the R console. For a short program, this method is quite convenient. This method could be used to debug our or others' R programs. We will discuss it further in chapter 3: "Black-Scholes-Merton Option Model."

Method 2: Using the source() Function

After we generated the above three lines using an editor such as Notepad and saved it (e.g., **c:/my_project/test.R**), now we can run the program using the function of **source()**.

```
> source("c:/my_project/test.R")
```

To view whether all three functions are available, we use the **ls()** function.

```
> ls()
[1] "fv_f" "pv_f" "pv_perpetuity"
```

The second way to use the **source()** function is to click **File** on the R menu bar. Go to **Source R codes...**, then locate your program. How about generating the following multiple-line program and saving it to a file called **c:/my_project/test02.R**?

```
pv_f<-function(fv,r,n) {
"Objective: estimate present value
fv : future value
r : discount rate
n : number of periods
e.g.,
> pv_f(100,0.1,1)
[1] 90.90909
";return(fv*(1+r)^(-n))
}
```

Below is the general procedure to call a prewritten R program.

```
# 2-step to run a R program
# [click] "file" ->"change dir..." -> [choose working directory]
# [click] "file" ->"Source R code"-> [choose your R program]
```

If we don't want to change our working directory, then we can simply include a path in our codes (i.e., using so-called absolute-path method).

```
> source("c/yan/w1_03.R") # 2nd way to run an R program
> source("c:/my_project/test_02.R") # 3rd way to run an R program
```

To view each step when executing, we add **echo=T**.

```
> source("c:/my_project/test_02.R") # echo=T: print each step
```

2.9. Using Meaningful Variable Names and Using the Tab Key Magically

For programming clarity, it is always a good idea to generate meaningful variables, such as pv for present value, fv for future value, pv_f for present value function, pv_annuity_f for present value function for annuity. By using those good names, we and other users would understand programs more easily. In addition, we can reduce unnecessary comments dramatically. Below is an example of using meaningful variables names.

```
data Mid3;
    set Mid3;
    DollarRealizedSpread_LR_SW=waDollarRealizedSpread_LR_SW/sumsize;
    DollarRealizedSpread_LR_DW=waDollarRealizedSpread_LR_DW/sumdollar;
    PercentRealizedSpread_LR_SW=waPercentRealizedSpread_LR_SW/sumsize;
    PercentRealizedSpread_LR_DW=waPercentRealizedSpread_LR_DW/sumdollar;
    DollarPriceImpact_LR_SW=waDollarPriceImpact_LR_SW/sumsize;
    DollarPriceImpact_LR_DW=waDollarPriceImpact_LR_DW/sumdollar;
    PercentPriceImpact_LR_SW=waPercentPriceImpact_LR_SW/sumsize;
    PercentPriceImpact_LR_DW=waPercentPriceImpact_LR_DW/sumdollar;
    DollarRealizedSpread_EOH_SW=waDollarRealizedSpread_EOH_SW/sumsize;
    DollarRealizedSpread_EOH_DW=waDollarRealizedSpread_EOH_DW/sumdollar;
    PercentRealizedSpread_EOH_SW=waPercentRealizedSpread_EOH_SW/sumsize;
    PercentRealizedSpread_EOH_DW=waPercentRealizedSpread_EOH_DW/sumdollar;
    DollarPriceImpact_EOH_SW=waDollarPriceImpact_EOH_SW/sumsize;
    DollarPriceImpact_EOH_DW=waDollarPriceImpact_EOH_DW/sumdollar;
    PercentPriceImpact_EOH_SW=waPercentPriceImpact_EOH_SW/sumsize;
    PercentPriceImpact_EOH_DW=waPercentPriceImpact_EOH_DW/sumdollar;
      DollarRealizedSpread_CLNV_SW=waDollarRealizedSpread_CLNV_SW/sumsize;
    DollarRealizedSpread_CLNV_DW=waDollarRealizedSpread_CLNV_DW/sumdollar;
    PercentRealizedSpread_CLNV_SW=waPercentRealizedSpread_CLNV_SW/sumsize;
    PercentRealizedSpread_CLNV_DW=waPercentRealizedSpread_CLNV_DW/sumdollar;
    DollarPriceImpact_CLNV_SW=waDollarPriceImpact_CLNV_SW/sumsize;
    DollarPriceImpact_CLNV_DW=waDollarPriceImpact_CLNV_DW/sumdollar;
    PercentPriceImpact_CLNV_SW=waPercentPriceImpact_CLNV_SW/sumsize;
    PercentPriceImpact_CLNV_DW=waPercentPriceImpact_CLNV_DW/sumdollar;
run;
```

The above codes are SAS codes. The complete codes could be downloaded at http://canisius.edu/~yany/longVariableNames.pdf. The name of the paper is "Liquidity Measurement Problems in Fast, Competitive Markets: Expensive and Cheap Solutions" by Holden and Jacobsen, a *Journal of Finance* paper.

Magic Use of the Tab Key

Assume that we have defined several meaningful names (see below). In chapter 1, we know that we could type a variable name to show its value. For instance, if we type pvAnnuity, we will get 100.

```
> pvAnnuity<-100
> pvPerpetuity<-200
> fvAnnuityDue=300
> pvAnnuity
[1] 100
```

Obviously, typing those long names is prone to error. Try the following:

```
> pvA # now we hit tab key to see the magic!
```

After hitting the **Tab** key, the complete name of **pvAnnuity** would pop up. The rule is that we just need to type enough letters to distinguish this variable from others; then we hit the **Tab** key. This is true when we type the **source()** command. Assume that we have a **pv_f.R** program located under **c:/yan/teaching/04_MGF690/pv_f.R** (# location c:/yan/teaching/04_MGF690/pv_f.R).

We can hit the **Tab** key several times to save time and effort or impress others.

```
> source("c:/yan/te # hit the tab key now
> source("c:/yan/teaching/ # we will see this one
> source("c:/yan/teaching/04 # type 2 integers, hit tab key
> source("c:/yan/teaching/04_MGF690/# see this
```

2.10. Changing Our Working Directory

To find out the current working directory, we use the getwd() function.

```
> getwd()
[1] "C:/Users/yyan/Documents"
```

It is always a good idea to generate a directory that contains all our data, programs, and other related materials for one specific project. After launching R, usually we want our working directory associated with the directory. The first way to change our working directory is to use the menu bar.

```
#[click] "File" - -> "Change dir…"
```

Alternatively we can use the **setwd()** function to change our current working directory. After it is done, we use **getwd()** to confirm.

```
> setwd("c:/temp") # change our current working directory
> getwd() # find out our current working directory
[1] "c:/temp"
```

2.11. Listing Files Under the Current Working Directory

The **dir()** function is used to list all programs, data sets, and other files under our current working directory.

```
> dir() # show all programs in the current working directory
```

Sometimes we want to pick up just a few files. In those cases, we specify a pattern by using **pattern='my_pattern'**.

```
> dir(pattern="ratio") # list files with "ratio" in their names
```

If we intend to check the files under another directory, add path="...". Again, this is called absolutepath method.

```
> dir(path="c:/temp/",pattern='test.R')
```

2.12. Default Value for an Input Argument

We can have default values for some or all of our input arguments.

```
> pv_f<-function(fv=100,r=0.05,n=1) fv*(1+r)^(-n)
```

Below shows how to call this function.

```
> pv_f() # use all default values
[1] 95.2381
> pv_f(fv=150) # use two default values
[1] 142.8571
```

With no default specified and we call the **pv_f** function without giving an appropriate input set, we would get an error message.

```
> pv_f<-function(fv,r,n) fv*(1+r)^(-n)
> pv_f()
Error in fv * (1 + r)^(-n) : 'fv' is missing
```

2.13. Comparison Between Two Listing Functions

We should not be confused between **dir()** and **ls()**. The former, **dir()**, lists files under our current working directory (or another directory if using the absolute-path method) while the **ls()** function lists all objects in our current working space (memory).

```
> ls() # list all objects
```

R objects include variables, lists, data frames, vectors, metrics, arrays, and functions. Don't worry if you have no clue about the meaning of *lists, data frames, metrics,* or *arrays*. We will discuss those in the later chapters.

```
> ls(pat='test') # show all objects contain 'test'
```

Another way to show all objectives is to use the function of **objects()**.

```
> objects() # 2nd way to show all objects
```

We can use the rm() function to remove unnecessary variables (objects) from our memory.

```
> rm(x) # remove x only
> rm(x,y) # remove both x and y
```

There are several ways to remove all objects.

```
# rm(list=ls((all=TRUE)) # remove all objects (method 1)
# rm(list=ls()) # a simpler version to remove all
# 2nd way to remove all objects
# [click] "Misc" -> "Remove all objects"
```

On the other hand, if we want to remove a file from our working directory, we have to delete it manually or issue the following command from the R prompt.

```
> file.remove('test.R') # relative path
> file.remove('c:/temp/test2.R') # absolute path
```

2.14. Grouping Many Small Functions into One File

When we have many small functions around a topic, it is a good idea to put them into a program. Then we issue one-line R codes to "activate" those functions.

There are several issues here. First, each individual program should be bug free. It is really time-consuming to debug multiple programs simultaneously. Second, add enough comments to each program. Third, arrange those programs in a logical order. Last but not least, we could add a nice header explaining the purpose of our set of programs.

```
"fin_101.R
Objective : a set of 50 programs related to Finance 101
Author : John Doe
Date : 10/2/2013
Modified : 7/3/2014
A list of all functions
1)     pv_f()
2)     fv_f()
3)     IRR()
4)     Bond_price()
5)     pv_annuity()
6)     pv_perpeturity()
"
# program one
# program two
# program three
```

If you are taking a financial-modeling course and using this book, it is a good idea to have a text file, such as **mgf690.R** or another name, for the whole course. There are several advantages for keeping such a text file. First, after you figure out a simple program, you can include it in this file. Second, since the file is in a text format, it is very easy to open, view, and modify. Third, many codes in the early chapters are quite useful for the later chapters. Thus, you can simply copy and paste relevant programs as your starting program. Fourth, it is quite easy to upload/activate your most used programs. Last but not least, by the end of the course, you should have a file that contains eighty R programs related to various functions. Those functions could be useful in the future.

2.15 Using R as a Financial Calculator

We could put all our finance functions written in R into a simple program and call it **fin_101.txt**. Then we can activate it by using **source("path/fin_101.txt")**, where *path* will be your specific path. Users could generate their own functions as well by following these steps.

Table 2.1. Steps to run **fin_101.txt**

Step	Description
1	Put all programs (functions) into a text file (e.g., **fin_101.txt**) and save the file to a specific location (e.g., **c:\temp\fin_101.txt**).
2	Launch R.
3	From R, click **File** then **Change dir** then **c:\temp**
4	Click **File** then **Source R codes**... then **fin_101.txt**

Now you are ready to call those functions included in **fin_101.txt** and use R as a financial calculator. Remember to use the **ls()** function to list all functions and type **pv_f** to view its usage. For step 4, the equivalent command is

```
> source('fin_101.txt')
```

If you don't want to change your working directory, you could combine steps 3 and 4 by issuing the following command:

```
> source('c:\\test_R\\fin_101.txt')
```

Or

```
> source("c:/test_R/fin_101.txt")
```

Table 2.2 below lists the advantages and barriers of using R in our introductory finance courses. Its flexibility means that users could adopt their own favorite function names. For instance, a user could rename **pv_f** as pv_function or **my_PV_function**. When an undergraduate student pursues a masters degree, the knowledge of R will give her/him a comparative advantage. The knowledge and skills of using R add certain weight when a graduate tries to land a Wall Street job since many financial institutions are using S-PLUS, a cousin of R. If you want to keep the original function, you can add another name instead.
See an example below.

```
> my_PV_function(fv,r,n) pv_f(fv,r,n)
```

In the above program, the new function of **my_PF_function()** is the same as our original **pv_f()** function.

Table 2.2. Advantages and barriers of using R in Finance 101

Advantages of Using R	
1	No cost (free downloading)
2	No black box (transparent in terms of formulae and logic)
3	More flexible than a financial calculator or Excel (For example, a user could generate their own functions by renaming the existing functions.)
4	Users could view examples for each function.
5	Could estimate market risk, total risk, liquidity measure, CAPM
6	Could download data from the Internet such as Yahoo Finance
7	Way more powerful than financial calculators and Excel
8	Extensible for many extrafunctionalities
9	Very useful for doing research
10	Good for a person's curriculum vitae
11	Used intensively in the financial industry
12	Many researchers around the world continue to develop R. More R packages would come.
13	More than three dozen packages are related to finance.
Disadvantages of Using R	
1	Most instructors don't know R.
2	It is easy to design a closed-book exam with a financial calculator.
3	Finance textbooks (in Finance 101) don't include R.
4	The current authors of finance textbooks might be reluctant to change their textbooks to incorporate R. If a textbook depends on both financial calculators and Excel, it is difficult for a student to learn another tool such as R. If a finance textbook is written with R as the primary tool of calculation, it will be well received by students.
5	The current publishers might be reluctant to change.
6	Resistance from the manufacturers of financial calculators
7	Mentality

2.16. Error Handling

Assume that we don't allow a negative interest rate (which means that you deposit your money in a bank and pay the bank the interest instead of the bank paying you).

```
> pv_f(fv=100,r=0.1,n=1)
[1] 90.90909
> pv_f(fv=100,r=-0.1,n=1) # accidently input a negative interest rate
[1] 111.1111
```

In these cases, we can use the **if-stop** function. See the following codes:

```
pv_f<-function(fv,r,n){
if(r<0)stop("r should be positive")
return(fv/(1+r)^n)
}
```