2
R FUNCTIONS

The focus of this chapter is on how to write simple R functions. To make first-time learners feel comfortable, we start with one-line functions for the present value of an annuity. We will discuss three ways to input variables. In addition, default values and how to make our functions self-explanatory will also be mentioned. In this chapter, the following topics will be covered:

- The simplest functions
- One-line function for the future value of one present value
- One-line function for the present value of one future value
- One-line function for the present value of a perpetuity
- One-line function for the present value of an annuity
- One-line function for the future value of an annuity
- One-line function for the present value of a growing annuity
- One-line function for the future value of a growing annuity
- Add comments and help to those functions.
- Three ways to input values into a function.
- Default values

2.1 WHAT IS A FUNCTION?

In programming and mathematics, a function is a named block of code or a mathematical expression that performs a specific task or computation. Functions take input values (arguments or parameters) and return a result, or perform actions based on those inputs. The primary purpose of functions is to encapsulate reusable pieces of code, making programs more modular, readable, and easier to maintain.

In mathematics, a function is a relation between a set of inputs (domain) and possible outputs (range), where each input is related to exactly one output. In programming, functions serve a similar purpose, allowing developers to break down complex tasks into smaller, more manageable units. Functions are defined using a name, a list of parameters,

and a body of code that specifies what the function does when called. When a function is called, the selected code block is executed, and the function may return a value or perform specific actions. Functions promote code reuse and abstraction, helping organize code into logical units and enhancing the clarity and efficiency of software development.

2.2 THE SIMPLEST FUNCTION

The most straightforward function takes just one line. Below, a function called '`dd`' is generated to double any input value.

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

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 part after will be the body of our function. To call it, treat it as any embedded R function, such as `min()` or `max()`.

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

We can use a more meaningful function name, such as `double_an_input`.

2.3 FUNCTIONS RELATED TO FINANCE

Most of the computations in R involve evaluating functions provided by R via packages written by professionals or individual users; we devote two chapters to discussing R packages (see Chapters 3 and 14). In this chapter, we focus on writing simple functions. After familiarizing ourselves with the basic structure, inputting values, and adding comments, we will use R as a financial calculator. In other words, we could write most finance-related functions in R using the 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 following chapters, we will explore this further by analyzing non-standard and more complex functions, such as the Black-Scholes-Merton option model.

2.4 USING MEANINGFUL FUNCTION NAMES

When writing financial functions, using meaningful names is a good idea. A meaningful name will make our program easier to read by reducing unnecessary comments. For example, if we want to write a function to calculate the present value of an annuity, we can use `pv_annuity()` as our function name. It is way better than just `my_function()`.

2.5 ONE-LINE FUNCTION FOR FINANCE

The most straightforward function has just one line. First, let's look at the future value of a given present value. The formula is given below.

$$FV = PV(1+R)^n, \tag{1}$$

where FV is the future value, PV is the present value, R is the period rate, and n is the number of periods. The one-line R code is shown below.

```
fv_f<-function(pv,r,n)pv*(1+r)^n
```

In the above code, `fv_f` is the function name, `function()` is the keyword, `pv`, `r`, and `n` are the inputs, and `pv*(1+r)^n` is the output. We can type the line above directly in the R console to activate it. Alternatively, we can open a text editor such as Notepad first, then copy the contents into the R console.

```
> fv_f(100,0.1,1)
  [1] 110
> fv_f(400,0.06,6)
  [1] 567.4076
```

We could also write a function that takes no inputs. For instance, type `q()` to quit R. If you type `exit()`, we will get an error message.

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

Thus, we could write a function called `exit()` that's equivalent to `q()`.

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

2.6 THREE WAYS TO INPUT VALUES

To call or run a function, we have three methods to input values (Positional, Keyword, or Mixed). The first method is easy to understand; the meaning of an input variable depends on its position. What is the present value of $100 that occurs in one year with a discount rate of 8%? Based on the related formula, we have the following one-line R code.

$$PV = \frac{FV}{(1+R)^n}, \tag{2}$$

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

According to the `pv_f()` structure, the inputs are in the order: 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)`, 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 must 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
```

The second method is called the "keyword approach": specify a keyword before 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, type its name, such as `pv_f`.

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

The last method of inputting variables is the so-called "Mixed approach." Knowledge of the two methods above should suffice. However, we introduce this method for completeness, as follows.

Rule 1: Exact math: match keywords from your input variables precisely the 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 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 could check our results with the following code.

```
# 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")
}
```

2.7 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, we show 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
```

We get an error message when no default value is specified, and we call the `pv_f` function without providing an appropriate input set.

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

There are some advantages to using a set of default values. The apparent reason is that we would see fewer error messages. Too many error messages discourage new learners from studying a new language. However, the disadvantage is that potential users may not know the predetermined default values.

2.8 MANY ONE-LINE FINANCE FUNCTIONS

Students will learn the following formulas for a few finance courses, such as Corporate Finance and Financial Management. The most straightforward formulas can be written as one-line R functions. Below, we list the formulas without much explanation.

$$PV = \frac{FV}{(1+R)^n} \quad , \tag{2}$$

$$PV(perpetuity) = \frac{c}{R} \tag{3}$$

$$PV(perpetuity\ due) = \frac{c}{R}(1+R) \tag{4}$$

$$PV(growing\ perpetuity) = \frac{c}{R-g} \quad , \tag{5}$$

$$PV(annuity) = \frac{c}{R}[1 - \frac{1}{(1+R)^n}] \tag{6}$$

$$FV(annuity) = \frac{C}{R}[(1+R)^n - 1] \tag{7}$$

$$PV(growing\ annuity) = \frac{c}{R-g}[1 - \frac{(1+g)^n}{(1+R)^n}] \tag{8}$$

$$FV(growwing\ annuity) = \frac{C}{R-g}[(1+R)^n - (1+g)^n] \tag{9}$$

In the above formulae, `FV` is the future value, `PV` is the present value, `R` is the discount rate, `n` is the number of periods, `c` is the amount of recurring cash flows, and `g` is a constant growth rate. For Equations (2) to (9), the first cash flow happens at the end of the first period. For detailed definitions of those functions, see Chapter 16: Finance Basics. Their corresponding one-line R programs are listed below.

```
pv_f<-function(fv,r,n)fv/(1+r)^n
#
pv_perpetuity<-function(c,r)c/r
#3
pv_perpetuity_due<-function(c,r)c/r*(1+r)
#
pv_growing_perpetuity<-function(c,r,g)c/(r-g)
#
pv_annuity<-function(c,r,n)c/r*(1-1/(1+r)^n)
#
fv_annuity<-function(c,r,n)c/r*((1+r)^n-1)
#
pv_growing_annuity<-function(c,r,n,g)c/(r-g)*(1-(1+g)^n/(1+r)^n)
#
fv_growing_annuity<-function(c,r,n,g)c/(r-g)*((1+r)^n-(1+g)^n)
```

2.9 PROGRAMS WITH MULTIPLE LINES

One-line R programs are best suited to simple formulas and tasks. Most functions have more than one line. For a multi-line function, a pair of curly braces, '{' and '}', encloses those lines.

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

We could still add a pair of curly braces around the simplest one-line functions to enclose the body of the main function.

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

2.10 TWO TYPES OF COMMENTS

The first type of comment starts with a #, shown below.

```
>pv=100 # present value
>r=0.1  # rate
>n=5    # number of periods
>fv<-pv*(1+r)**n
print(fv)
161.05100000000004
```

We can use the round() function to round to a specified number of decimal places.

```
fv2=round(fv,2)
print(fv2)
161.05
```

The second type of comment, shown below, consists of two pairs of double quotations. Below, it is suitable for multiple-line comments rather than single-line ones.

```
"
  second comments

  line 1
  line 2

  line n

"
```

In the next section, we will use the second type of comment to add help or comments to our one-line R functions, making them self-explanatory.

2.11 WRITING OUR FUNCTION WITH COMMENTS

The best strategy for making our functions self-explanatory is to add a few comments, such as the program's objective, definitions of input variables, and one or two examples of how to apply the function.

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

In the above program, for the last line before the second curly brace, we use "`return()`." This is a standard command for returning our final result. Adding '`return()`' makes our programs clearer, even if its omission causes no errors. 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 compiler ignores comments when compiling the program. Using many number signs for a multi-line comment is cumbersome because we must add one before each comment line. In those cases, a pair of double quotation marks, " and ", is used to enclose all comment lines; see the example above.

Below is another example:

$$fv = pv(1+R)^n ,$$

Where `pv` is the present value, `fv` is the future value, `R` is the discount rate, and `n` is the number of periods. For example, if we deposit $100 in a bank today for 3 years at an annual rate of 2%, what is the future value? The answer is $106.12, shown below.

```
100*(1+0.02)**3
Out[37]: 106.12080000000002
```

We can write two lines of code for Equation (1), shown below.

```
fv_f<-function(pv,r,n){
    "Objective   : calculate the future value
    formula used : fv = pv(1+r)^n
              pv : present value
               r : periodic rate
```

```
                n : number of periods
    Example #1: fv_f(100,0.02,3)
                104.03999999999999

    Example #2:fv=fv_f(100,0.02,3)
               fv2=round(fv,2)
               print(fv2)
               106.12
  "
  fv=pv*(1+r)**n
  return(fv)
}
```

The beauty is that we can type `fv_f` to see our comments, as shown below.

```
> fv_f
function(pv,r,n){
  "Objective   : calculate the future value
    formula used : fv = pv(1+r)^n
              pv : present value
               r : periodic rate
               n : number of periods
      Example #1: fv_f(100,0.02,3)
                  104.03999999999999

      Example #2:fv=fv_f(100,0.02,3)
                 fv2=round(fv,2)
                 print(fv2)
                 106.12
    "
  fv=pv*(1+r)**n
  return(fv)
}
>
```

Figure 2-1 The complete code for the future value function (`.fv_f`)

The above output would tell other users the formula, define the three input variables, and provide two examples.

2.12 WELL-INDENTED CODE IS MORE READABLE

Unlike Python, where indentation is critical, spaces and indentations are not crucial for R. However, a well-indented R program will be more readable. The following two programs are essentially the same except for indentation. 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 code.

```
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(1+r))
}
```

A good indentation's effectiveness is less evident for a simple function, such as the example above. However, proper indentation is critical for more complex programs, such as those with multiple loops and blocks.

2.13 MEANINGFUL NAMES/THE MAGIC TAB KEY

For programming clarity, generating meaningful variables, such as `pv` for present value, `fv` for future value, `pv_f` for present value function, and `pv_annuity_f` for present value function for annuity. By using those good names, other users and I would understand programs more efficiently. In addition, we could dramatically reduce unnecessary comments. Below is an example of using meaningful variable 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;
```

Figure 2-2 An illustration using meaningful function names

The above codes are SAS codes. The complete codes can be downloaded at http://datayyy.com/doc_pdf/longVariableNames.pdf. We can use the Tab key to complete

a long variable or function name automatically. Assume that we have defined several meaningful names; see below. In Chapter 1, we learned that we can type a variable name to display its value. For instance, if we type `pvAnnuity`, we will get 100.

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

It is prone to errors when typing those long names. Try the following one.

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

After pressing the Tab key, the full name of `pvAnnuity` would pop up. The rule is to type enough letters to distinguish this variable from others, then hit the Tab key.

2.14 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 create a directory to include all our data, programs, and other related materials for one specific project. After launching R, we usually want the working directory to be associated with that directory. The first way is to use the `setwd()` function to change our current working directory. After it is done, we use `getwd()` to confirm. One example is given below.

```
> setwd("c:/test_R") # change our current working directory
> getwd()            # to confirm
  [1] "c:/test_R"
```

Alternatively, we can use the menu bar shown below.

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

2.15 LISTING FILES: `DIR()`

The `dir()` function lists all programs, datasets, and files in the current working directory.

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

Sometimes we want to pick up 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=my_path". Again, this is called the absolute-path method.

```
> dir(path="c:/test_R/",pattern='test')
```

2.16 COMPARISING ls() AND dir()

We should not confuse ls() with dir(). The former, ls(), lists all objects in our current working space (memory), while dir() lists files under our current working directory (or another directory if using the absolute path method).

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

R objects include lists, data frames, vectors, matrices, arrays, and functions. You are fine if you do not know the meanings of lists, data frames, matrices, or arrays. We will discuss those in the later chapters.

```
> ls(pattern='test') # show all objects containing 'test'
> ls(pat='test')     # same as the above
```

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

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

The rm() function removes unnecessary variables or 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:/test_R/test2.R')   # absolute path
```

2.17 USING NOTEPAD AS A TEXT EDITOR

There are several ways to initiate Notepad.

```
# Method 1: [click] "start" - -> [click] notepad
# Method 2: [click] "start" - ->  enter "notepad"
```

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

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

The image is shown below.

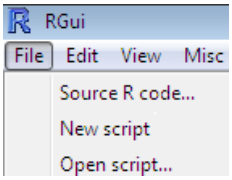

Figure 2-3 Using an R editor (R console)

In addition, we can write or edit our programs using WordPad, MS Word, or other word editors. Just remember to use text format when saving our programs. For WordPad, the following image shows the file format (R program).

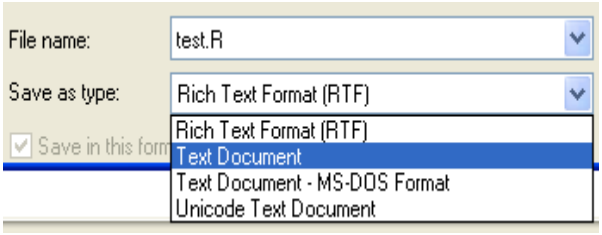

Figure 2-4 When saving an R program

For MS Word, we must export our R program as a text file, not a docx file, as shown below.

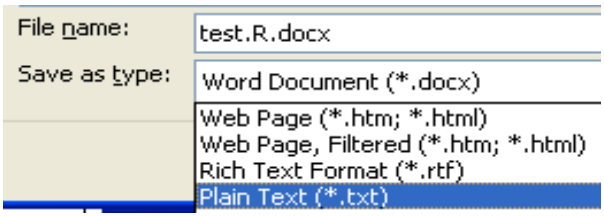

Figure 2-5 The docx is an MS Word extension

2.18 PROGRAM EXTENSIONS ARE NOT CRITICAL

When writing an R program, its extension is not essential. For example, we could name it `test.txt`, `test.R`, or just `test`, i.e., completely ignore the extension. The advantage of a '.txt' extension is that our computers will automatically launch Notepad when we click it. Similarly, if we intend to use the R editor to open our programs, adopting an "`.R`" extension is a good idea. Another advantage of using `.R` as an extension is that it distinguishes our programs from other files, such as input, output, or data files, which use a '.txt' extension.

2.19 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 I: "Copy and paste"
First, we use Notepad to generate those three lines. Then, highlight them and paste them into 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 R programs.

Method II: using the `source()` function
After we generate the three lines above using an editor such as Notepad, we save them in a directory, e.g., `c:/my_project/test.R`. To run the program, we use the function `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, "Source R codes…", then locate your program. Try to generate the following multiple-line program and save 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 for calling a pre-written R program.

```
# 2-step to run an 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, we can include a path in our code, i.e., using the so-called absolute-path method.

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

To view each step as it executes, we add the "echo=T" parameter.

```
> source("c/yan/w1_03.R", echo=T) # echo=T: print each step
```

We can use the `dir()` function to list our variables and functions.

```
>>>futureValue=125.4
```

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. We could press the Tab key several times to save time and effort, or to 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
```

2.20 RUNNING PROGRAMS WITH RSTUDIO

To start a new file, click on "File" on the menu bar, then "New File," then choose "R Script Ctrl+Shift+N," as shown below.

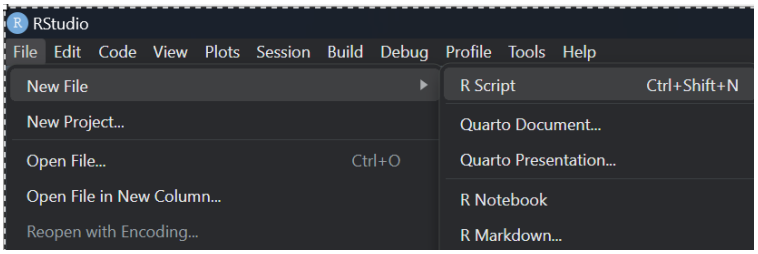

Figure 2-6 Choosing "`New File`", then "`R Script`"

Then we can enter our program, such as `pv_f<-function(fv,r,n)fv/(1+r)^n`. To run a program, highlight the program, then hit Ctrl-End.

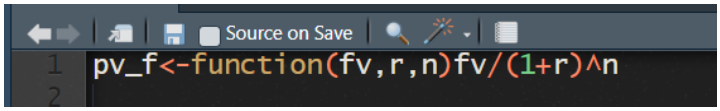

Figure 2-7 One-line present value function

The above image will appear in the bottom R-console.

2.21 GROUPING SMALL FUNCTIONS TOGETHER

When we have many small functions around a topic, putting them into a program is a good idea. Then, we issue one-line R codes to "activate" those functions. There are several issues here. First, each program should be bug-free. Debugging multiple programs simultaneously is time-consuming. Second, enough comments needed to be added to each program. Third, those programs should be arranged in a logical order. Last but not least, we could add a nice header explaining the purpose of our set of programs.

```
fun_101<-function(i){
" fin_101.R
  Objective: a set of 50 programs related to Finance 101
  Author: John Doe
  Date      : 10/2/2013
  Modified  : 1/3/2024

  A list of all functions
         .1)  pv_f()
         .2)  fv_f()
         .3)  pv_perpetuity
         .4)  pv_annuity
         .5)  pv_annuity_due
         .6)  fv_annuity
         .7)  fv_annuity_due
         .8)  pv_growing_annuity
         .9)  fv_growing_annuiay
        .10)  IRR()
        .11)  Bond_price()
        .12)  NPV_f()
        .13)  PaybackPeriod()
"
      # program one
      # program two
      # program three

      # more here

}
```

When taking a financial modeling course using this book, it is a good idea to have a text file, such as `mgf690.R`, for the entire course. There are several advantages to keeping such a text file. First, once you have a simple program, you can include it in this file. Second, since the file is in a text format, it is easy to open, view, and modify. Third, many of the codes in the early chapters are pretty helpful in the later chapters. Thus, you can copy and paste relevant programs as your starting program. Fourth, it is pretty easy to upload/activate your most used programs. Last but not least, by the end of the course, you should have a file containing 80 R programs covering various functions. Those functions could be helpful in the future.

2.22 USING R AS A FINANCIAL CALCULATOR

We could put all our finance functions written in R into a simple program called `fin_101.txt`. Then, we can activate it by using `source("path/fin_101.txt")`, where "`path`" will be your specific path. Users could also generate their functions by following the steps below.

Table 2.1 Steps to run fin_101.txt

Step	Description
1	Put all programs (functions) into a text file, e.g., call it fin_101.txt, and save the file to a specific location, e.g., c:\test_R\fin_101.txt.
2	Launch R
3	[from R] click File - -> Change dir - -> c:\test_R\
4	[click] File - - > Source R codes …. - -> 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 one of the following commands.

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

Or

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

Table 2.2 lists the advantages and barriers of using R in our introductory finance courses. Flexibility means that users can adopt their favorite function names. For instance, a user could rename `pv_f` as `pv_function`, or `my_PV_function`. When an undergraduate student pursues a master's degree, the knowledge of R will give them a comparative advantage. The knowledge and skills of R add a 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 the example below.

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

In the program above, the new function my_PF_function() is identical to our original `pv_f()` function.

2.23 ERROR HANDLING

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

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

We could use the `if-stop` function; see the following code.

```
pv_f<-function(fv,r,n){

     if(r<0)stop("r should be positive")

     return(fv/(1+r)^n)

}
```

For a complex program, we can generate many 'marks', such as `print("pass A")`, `print("pass B")`, and the like.

2.24 THE PRECISION OF R

The precision of our R software is not an issue for most researchers or calculations. However, knowing how to find it would be helpful if you have such an issue in the future.

```
> .Machine$double.eps
[1] 2.220446e-16
```

REFERENCES

Holden and Jacobsen, 2022, "Liquidity Measurement Problems in Fast, Competitive Markets: Expensive and Cheap Solutions," The Journal of Finance, http://datayyy.com/doc_pdf/longVariableNames.pdf.

Appendix A: Advantages and barriers of using R in Finance

Panel A: Advantages of using R	
1	No cost (free downloading)
2	Not a black box (transparent in terms of formulae and logic)
3	It is more flexible than a financial calculator or Excel. For example, users could generate their functions by renaming existing functions.
4	Users could view examples for each function.
5	Could estimate market risk, total risk, liquidity measure, and CAPM
6	Could download data from the internet, such as Yahoo Finance

7	Way more potent than financial calculators and Excel
8	Extensible for many extra functionalities
9	Very useful for doing research
10	Suitable for a person's curriculum vitae
11	R is used intensively in the financial industry.
12	Many researchers around the world continue to develop R so that more R packages will come.
13	More than three dozen packages related to finance
Panel B: Barriers to using R	
1	Most instructors don't know R.
2	It is easy to design a closed-book exam with a financial calculator.
3	No finance textbook (in Finance 101) includes 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, students will receive it well.
5	The current publishers might be reluctant to change.
6	Resistance from the manufacturers of financial calculators
7	Mentality

Appendix B: After launching R, issue one of the following lines of R code.

```
source("http://datayyy.com/fmr/week1.txt")
```

```
*-------------------------------------------------------*
* Financial Modeling using R (2nd edition)  2026 Yan  *
*-------------------------------------------------------*
*  .c1  R Basics                                        *
*  .c2  R functions                                     *
*-------------------------------------------------------*
* >.c2       # go to chapter 1 (a dot in front of c1) *
* >.uu       # go to utility submenu                   *
* >.fm       # back to this menu                       *
*-------------------------------------------------------*
```

Figure 2-8 The menu for Week 1

Appendix C: For Chapter 2, type `.chapter2` or `.c2`, as shown below.

```
> .chapter2
function(i=0){
" i Chapter 2: Functions
  - ----------------------------------------
  1  What is a function?
  2  The simplest function
  3  PV function for one given future cash flow
  4  Present value formula for perpetuity/annuity
  5  Perpetuity due and annuity due
  6  R editor
  7  Three ways to input values
  8  Two types of comments
  9  Future value of one present value
 10  How to activate an R function
 11  Magic use of the tab key
 12  Show and change our working directory
 13  Definition of NPV and NPV rule
 14  Files under the working directory
 15  Function with default values
 16  Definitions of IRR and IRR rule
 17  Well-indented code is more readable
 18  Total risk and standard deviation
 19  Videos
 20  Links

 Example #1:> .c2    # get the above list
 Example #2:> .c2()  # get the above list
 Example #3:> .c2(1) # see the first explanation
```

Figure 2-9 The contents of Chapter 2

Appendix D: Changing the appearance of RStudio.
Click on "Tools" on the menu bar, then "Global Options," then choose 'Appearance,' as shown below.

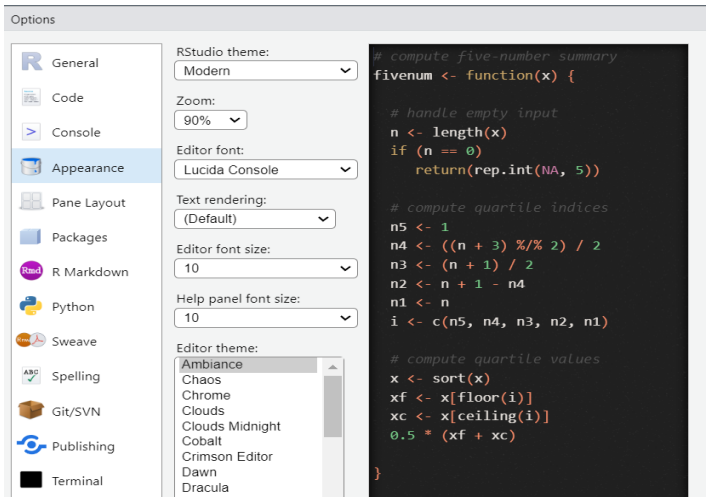

Figure 2-10 Changing the setting of RStudio

QUESTIONS

2.1 What are the critical components of an R function?

2.2 Write a new R function called `std()`,the same as the embedded R function called `sd()`. By the way, the `sd()` function estimates the standard deviation of an input variable (a vector or matrix).

2.3. Explain the following function

```
> echo<-function(x) print(x)
```

2.4 Write an R function to estimate the present value of perpetuity where `c` is the cash flows, and `r` is the discount rate. The first cash flow happens at the end of the `n`th period.

2.5 Write an R program to estimate IRR (Internal Rate of Return) for a given set of cash flows. You can assume the first cash flow occurs today.

2.6 A project contributes a cash inflow of $5,000 at the end of the first year and $8,000 at the end of the second year. The initial cost is $3,000. The appropriate discount rates are 10% and 12% for the first and the second years, respectively. What is the present value of the project?

2.7 Write an R function to estimate the following sum.

$$S = a_1 + a_1 q + a_1 q^2 + a_1 q^3 + \ldots + a_1 q^n + a_1 q^{n+1} + \cdots$$

$$S = \frac{a_1}{1-q} \qquad q < 1$$

2.8 Modify your code above (2.7) to disallow situations where $q \geq 1$.

2.9 Write an R program to calculate the present value of a growing perpetuity.

2.10 Modify the above program to calculate the growing perpetuity due (defined as all cash flows happening at the end of the periods).

2.11 What is the effective annual rate for a 10% annual rate compounding semi-annually?

2.12 Write an R function to estimate the effective annual rate for a given APR (Annual Percentage Rate) compounding `m` times per year; the formula is R=(1+APR/m)^m-1.

2.13 You have just noticed on the financial pages of the local newspaper that you can buy a bond for $750. If the coupon rate is 10%, coupon payments are semi-annual, and there are ten years to maturity, should you purchase it if your required return on investments of this type is 13%?

2.14 Write an R function called `pv_bond()` to estimate the price of a bond given `c` (coupon payment), `R` (discount rate), `P` (principal), and `n` (the number of periods).

2.15 If a firm's earnings per share grow from $1 to $2 over seven years, what is the annual growth rate?

2.16 If the annual rate is 9.5%, compounded quarterly, what is the equivalent rate compounded continuously?

2.17 Write an R program to estimate a continuously compounded rate from another rate, $R_{cont} = m * \ln\left(1 + \frac{R_m}{m}\right)$, where m is the number of compounding periods per year.

2.18 The continuously compounded rate is 8.34%. What is the corresponding annual rate, compounded semi-annually?

2.19 Write an R program to estimate $R_m = m(e^{\frac{R_c}{m}} - 1)$, where `Rm` (annual rate compounded *m* times per year) for a given rate compounded continuously (R_c).

2.20 Combine at least ten essential finance-related functions, such as `pv_f` and `fv_f`, to generate a file called `fin_101.txt`.

2.21 Write an R function to calculate BMI (Body Mass Index), defined as Weight in Kg divided by squared height (in meters).

$$BMI = \frac{W}{H^2} = \frac{kg}{m^2}$$

There are two inputs: 1) Input Kg and m directly, and 2) input foot (or inch) and pound. Modify the program further to include the following information: show the text based on their corresponding ranges after offering a BMI value.

Description	BMI
Underweight	<18.5
Normal	18.5 to 25
Overweight	25 to 30
Obese	>30

2.22 (This is an open-ended question.) Write a financial calculator with as many functions as possible.