Mathematics of Finance, Courant Institute, Spring 2019
https://www.math.nyu.edu/faculty/goodman/teaching/MathFin2019/MathFinance.html

Logic, programming practice and style

This is midway trough the quick introduction to `R`. You should test your understanding by typing in the examples here and changing them in small ways. There are three "keys" to being good at coding:

- Understand the programming language as deeply as you can. That means environments, argument passing, objects (variables), etc. The "for dummies" approach of copying examples from a book will make you a bad coder.
- Always follow rules of software practice and coding style.
- Practice.

Reserved words

A *reserved word* is a word that is part of the language and cannot be used as a variable (object) name. Suppose, for example, you want variables `df` for "domestic funds" and `if` for "international funds". Here's what happens when you try that at the command line:

```
> df = 4
> if = 5
Error: unexpected '=' in "if ="
>
```

The word `if` (the English word "if", see below) is a reserved word that cannot be used as the name of an object. You can get a list of all reserved words by typing `help(reserved) [enter]` at the `R` command prompt. You don't have to memorize the list. You will soon know most of the words on it just from using `R`.

Every word in the reserved-word list is there for a reason. Often, it's because the word tells the `R` interpreter something about what kind of command it is. For example, the reserved word `function` tells the interpreter that you are defining a function. If `function` were the name of something, then the command

```
> f = function(x) { . . .
```

would be ambiguous. It could mean either that `f` is a variable whose value is `function(x)`, or it could mean that `f` itself is a function. There are different kinds of commands: assignments, expressions, definitions, etc. The interpreter uses reserved words to tell what kind of command it is. Several reserved words, including `if` and `TRUE`, are described below. It should be clear in each case why the word has to be reserved. This explains the error message: `Error:`

`unexpected '=' in "if ="`. As will be clear below, what comes after `if` must be a logical expression, not an `=` sign. The `=` is *unexpected*.

Programming languages also have *keywords*. This is not the same as reserved words, but many people (including me) often say "keyword" when we should say "reserved word".

Logical variables and expressions

An *object* (also called *variable*) in `R` can have *type* `logical`. Other types that we have seen are `double` and `integer` (two kings of number), and `character` (a character string). A logical variable can have only two values, `TRUE` or `FALSE`. Here, we assign the object (variable) called `x` the value `TRUE`. The `R` interpreter then gives `x` the type `logical`.

```
> x = TRUE
> typeof(x)
[1] "logical"
```

A *comparison* expression has a logical value (`TRUE` or `FALSE`). Comparisons test whether two numbers are equal, or whether one is larger or smaller than the other. The comparison expression `( x > y ) [enter]` evaluates to `TRUE` if $x > y$ and evaluates to `FALSE` if $x < y$ or $x = y$. You test for equality (is it true that $x = y$?) using two "equals" signs, as `==`, not one. The "equals" sign in `R` (and most other languages) is assignment. But you can't assign 3 a value, so this command produces an error message. The $\leq$ test is `<=` and the $\geq$ test is `>=`. Any comparison expression evaluates either to `TRUE` or to `FALSE`.

```
>  ( 2 > 3 )
[1] FALSE
> ( 2 < 3 )
[1] TRUE
> ( 3 > 3 )
[1] FALSE
> ( 3 = 3 )
Error in 3 = 3 : invalid (do_set) left-hand side to assignment
> ( 3 == 3 )
[1] TRUE
> ( 3 >= 3 )
[1] TRUE
> ( 2 <= 3 )
[1] TRUE
>
```

There are more comparison operations than these. There are comparison operators for character strings. A web search will tell you how to do that kind of thing.

Besides comparisons, there are *logical operations*, called *not*, *and* and *or*. If x is a logical variable, then (not x) is the opposite of x. If x is true, then (not x) is false, and conversely. Not, in `R` is `!`. Here is an illustration of this:

```
> x = ( 3 > 2)
> x
[1] TRUE
> !x
[1] FALSE
```

The logical operation *and* is `&` in `R`. If `x` and `y` are logical variables, then `x & y` evaluates to `TRUE` if `x` and `y` are both `TRUE`. Otherwise `x & y` evaluates to `FALSE`. The logical operation *or* is `|` in `R` (a vertical bar). The expression `x | y` evaluates to `TRUE` unless both `x` and `y` are FALSE. Here is an illustration

```
> x = TRUE
> y = TRUE
> z = FALSE
> x&y
[1] TRUE
> x&z
[1] FALSE
> x|y
[1] TRUE
> x|z
[1] TRUE
> z|z
[1] FALSE
```

You can make complicated expressions with comparisons and logical operations. For example, this tests whether $x \in [0, 1)$. Note that $x = 1$ is not allowed but $x = 0$ is allowed. In logical expressions like this *always* use parens like this. It is very easy to get the wrong result because `R` precedence rules are weird. *Always* use parens for logical expressions (see below).

```
> x = 1.5
> ( x >= 0 ) & ( x < 1 )
[1] FALSE
```

Conditional execution, *if* test

Conditional execution means that script may or may not execute a command, depending on some test. You can do this in `R` with a command that starts with an *if test*. (We will see other ways.) It looks like:

```
if ( [logical value] ) { ... [code block] ...}
```

If the logical value in the if test is TRUE, then the commands in the code block are executed. If the logical value is FALSE, then the code block is skipped. If it's a script (it usually is), the R interpreter executes the next command after the close curly that ends the code block.

Figure 1 has an example, which is a function that returns $\sqrt{|x|}$. The first command in the function definition is an if test. The comparison `x < 0` evaluates to TRUE if $x < 0$. If it evaluates to true, then the code block is executed. In this case, it just replaces x with $-x$. That way, if x is negative, it is replaced with $-x$, which is $|x|$. If $x \geq 0$, the code block is not executed, and x stays x, which is $|x|$. Either way (TRUE or FALSE), the next command executed is `y = sqrt(x)`.

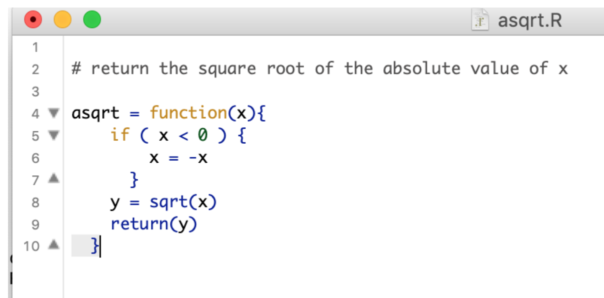

Figure 1: The R editor window where the `asqrt()` function is defined

Here is an illustration of `asqrt()` in action.

```
> source("asqrt.R")
> T = 4
> asqrt(T)
[1] 2
> T = -4
> asqrt(T)
[1] 2
> T
[1] -4
```

The first command reads the definition of `asqrt()` into the environment. The third command passes the value of T to this `asqrt()` function. Inside the body of the function (the curly braces that enclose the function commands) that value will be called x. Our `asqrt()` gives the right answer, which is 2. Next, we take `asqrt(T)` with $T = -4$. In this case, the function body of `asqrt()` replaces x with $-x$. But the value of T is not changed (see the last two lines). Argument passing in R is `call by value`. This means that the outside value of the calling argument (which is called T outside) is copied into a *local variable* that exists only in the environment of the function body. For that reason, changing the value of the local variable x does not change the outside variable T.

The function `asqrt()` illustrates that code blocks can be *nested*. This means that one code block can contain another code block. (A code block is a "nest" that can hold other code blocks?) In Figure 1, there is a code block that begins with an open curly on line 4 and ends with the close curly on line 10. This is the *outer* block. Inside this code block is a *nested* code block that starts with the open curly on line 5 and ends with the close curly on line 7. This is the *inner* block. Code blocks can be nested more than this, with inner blocks having inner-inner blocks and so on.

While **loops**

A *loop* is a code block that is executed repeatedly, until it is done. It is called *loop* because of the way it looks in a *flow chart* (Do a web search to see what flow charts look like.) The `do` loop we saw before is for situations where you know in advance how many *trips* through the loop you need. The `do` loop does one trip through the loop for every data value in an array. The `while` loop is for situations where you don't know in advance how many trips through the loop you will need. The while loop continues "looping" as long as some logical variable is `TRUE` or some expression evaluates to `TRUE`. It looks like

```
while ( [logical expression] ) { ... [code block] ... }
```

The `R` interpreter evaluates the logical expression. If it's `TRUE`, it executes the commands in the code block. This is like `if`. But then it evaluates the logical expression again. If it evaluates to `TRUE` again, it evaluates the code block again. This continues until the logical expression evaluates to `FALSE`.

Figure 2 has an example. It answers the question: how many years does it take to multiply an investment by a factor M with simple interest rate r per year? It assumes ("without loss of generality") that the initial investment is 1. It keeps multiplying the value V by $(1+r)$ until it exceeds the target M. Then it stops and returns V. Lines 5 and 6 are initializations. Note the comments that explain the roles of the variables. Line 7 starts the `while` loop. The loop is executed as long as $V < M$, which means that the value after T years is less than the target. The first command in the loop *body* formats a string for output. The number T (the number of years) is printed in `d` format, which we haven't seen until now. This prints an integer ("d" is for "decimal"?) with no decimal point, which is how you want the number of years to look. It prints the value in the `f` format we saw in the first handout. Lines 10 and 11 *update* T and V for the passage of one year. This process will continue, with V steadily increasing each trip through the loop, until the logical expression `( V < M )` evaluates to `FALSE`.

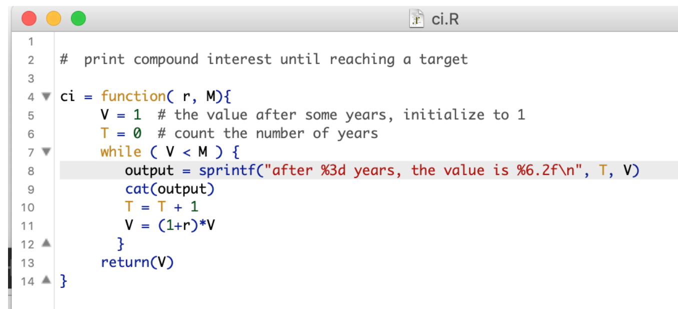

Figure 2: The `R` editor window where the `ci()` function is defined

Here's what happens when we try it at the command line. As always, we first have to read the definition of `ci()` into the environment. Then we use the function with $r = .05$ (which is 5% interest per year) and $M = 2$, which asks how many years it takes to double your money. The first time through the loop, $T = 0$ and $V = 1$. Then (though this is "silent" for now) V is replaces with $(1 + r)V$ and T is replaced with $T + 1$ (we say T is *incremented*). The next trip through the loop we have $T = 1$ and $V = (1 + r) = 1.05$. The body of the loop sets V to $(1 + r)^2 = 1 + 2r + r^2 = 1 + .1 + .0025 = 1.1025$. The test that controls the `while` loop still evaluates to `TRUE` because $1.1025 < 2$. The loop keeps executing and V slowly increases until the 14th through the loop. After printing, V gets the value 2.079. With this value, $V < M$ evaluates to `FALSE` and the looping stops.

The advantage of formatted output is that these numbers line up and are easy to read. You can see the year in `d` format without a decimal point and V in `f` format with a decimal point and two digits after the decimal point. If you want to know V more accurately, you could ask for 4 digits after the decimal point by replacing `%6.2f` with `%8.4f`. You have to increase the total number of characters from 6 to 8 (two more characters) when you ask for two more digits after the decimal point.

```
> source("ci.R")
> ci(.05, 2)
after   0 years, the value is   1.00
after   1 years, the value is   1.05
after   2 years, the value is   1.10
after   3 years, the value is   1.16
after   4 years, the value is   1.22
after   5 years, the value is   1.28
after   6 years, the value is   1.34
after   7 years, the value is   1.41
after   8 years, the value is   1.48
after   9 years, the value is   1.55
after  10 years, the value is   1.63
after  11 years, the value is   1.71
after  12 years, the value is   1.80
after  13 years, the value is   1.89
after  14 years, the value is   1.98
[1] 2.078928
```

Sometimes you don't find until the end of the loop body whether you're looped enough. There's a natural example of this in assignment 3. One way to handle this is to make a logical variable called, say, `continue`. You set `continue = TRUE` before going into the `while` loop. You put an `if` test at the bottom of the loop body to see whether you've looped enough. If you have looped enough, set `continue = FALSE`. Figure 3 has the loop done this way. At line 10, the `continue` variable is initialized to `TRUE`. Line 17 has an `if` test to see whether the loop *termination criterion* is satisfied. The termination criterion is that the loop should stop with $V > M$. If the criterion is satisfied, you get some more output (lines 19 and 20) and then `continue` is set to `FALSE` (line 21). This stops the loop because the next time line 10 is evaluated, the logical expression is `FALSE`.

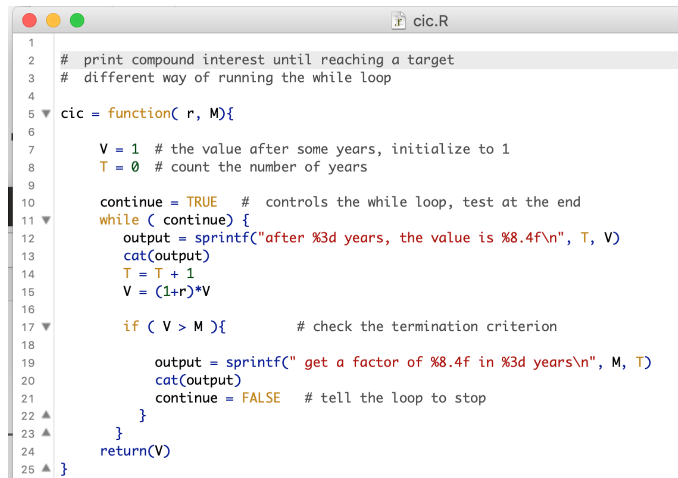

Figure 3: The R editor window where the `cic()` function is defined

Here's what happens when you run this. Notice that (see line 12 and line 19), we've asked for four digits instead of 2. This shows that $(1+r)^2$ is actually 1.1025, as the algebra above predicts (never trust algebra not confirmed by computation).

```
> source("cic.R")
> cic(.05,2)
after   0 years, the value is   1.0000
after   1 years, the value is   1.0500
after   2 years, the value is   1.1025
after   3 years, the value is   1.1576
after   4 years, the value is   1.2155
after   5 years, the value is   1.2763
after   6 years, the value is   1.3401
after   7 years, the value is   1.4071
after   8 years, the value is   1.4775
after   9 years, the value is   1.5513
after  10 years, the value is   1.6289
after  11 years, the value is   1.7103
after  12 years, the value is   1.7959
after  13 years, the value is   1.8856
after  14 years, the value is   1.9799
 get a factor of   2.0000 in  15 years
[1] 2.078928
```

Programming practice, programming style

Programming practice and programming style refer to things you do or don't do in order to make your code more likely to be correct and more useful to others in your team. Bad programming practice (examples below) make it more likely that your code has subtle bugs (mistakes) that can be time consuming to find. Bad programming style can make your code much harder for others to understand and modify. Even if you're on your own (a team of one), bad programming style can build up to make a code that is frustrating to you. Good programming practice and programming style asks the coder to spend a few more minutes when writing the code for the first time. The time saved in the long run usually makes this worthwhile.

In this class, you will be asked to hand in your code and output for each assignment. You will lose points on the homework if the person grading it doesn't like the programming practice or style. In a technical job interview, you can easily lose a job offer if your code looks bad. People out there developing software in the real world have all had the team-mate from hell who writes code that seems to be technically correct but in fact either has subtle bugs or is very time consuming to understand. They won't want you on their team.

Now would be a good time to read the `R` style guide posted on the `resources` page of the class web site. Here are some rules of style that you should keep in mind when coding.

- Put comments at the top of any function saying what it does
- Use *white space* (spaces or blank lines) to make the structure of the code clearer. Figure 3 as lots of blank lines to make the control lines (line 5, line 11, line 17) easy to spot.
- Indent the commands in any code block. The official `R` indentation is four spaces, but you can use anything between 2 and 5.
- Put the close curly on its own line. This also helps code blocks stand out.
- Comment any variable with something helpful. Example of an unhelpful comment: `T = 0 # set T to zero`.
- Make output easy to read – nicely lined up in columns, an intelligently chosen number of digits. Output like `L = 3.141592653589793` looks silly in most situations.
- Use "defensive programming".
- No *hard wired* constants (see "Infinite loops" for an example).

Defensive programming

Defensive programming (my term) means keeping in mind the things that can go wrong and doing something about them. This won't always prevent bugs,

because every program has bugs. But it can make bugs less destructive and easier to find. There's a TV comedy image of a nerd who constantly imagines disasters that are about to happen. People like that make good programmers.

Argument range checking

The function in Figure 3 has an arguments r that must be positive and M that must be larger than one. *Argument checking* could involve putting an `if` test at the top of the function and printing an error message if $r \leq 0$ or $M \leq 1$. The error message should be informative, so that you have some idea what went wrong when you see it. The error message should say the name of the function (`cic()`), the problem ($r \leq 0$) and the value of r ($r = -2$ in this case).

Infinite loops

A `while` loop is *infinite* if it never stops or if it takes too long to stop. For example suppose you give $r = -.1$ to the `cic()` function of Figure 3. The number V starts at $V = 1$ and keeps getting multiplied by $1+r = .9$. Multiplying by .9 will never make $V \geq M = 2$, so the loop continues forever. There's a panic that sets in when you run your program and realize it's in an infinite loop. How do you stop it? Do you have to quit the `R` app and restart it?

```
> cic(-.1, 2)
after   0 years, the value is   1.0000
after   1 years, the value is   0.9000
after   2 years, the value is   0.8100
after   3 years, the value is   0.7290
after   4 years, the value is   0.6561
after   5 years, the value is   0.5905
after   6 years, the value is   0.5314
          .
          .
          .
```

Figure 4 is a screen shot of the `R` command line console as it looks on my imac. It kept going, multiplying by $1 + r = .9$ each time, until I clicked on the red `STOP` sign in the top left of the `R` console.

```
R Console
~/Desktop/notes/MathFin2019/codes
> cic(-.1, 2)
after   0 years, the value is   1.0000
after   1 years, the value is   0.9000
after   2 years, the value is   0.8100
after   3 years, the value is   0.7290
after   4 years, the value is   0.6561
after   5 years, the value is   0.5905
after   6 years, the value is   0.5314
after   7 years, the value is   0.4783
after   8 years, the value is   0.4305
after   9 years, the value is   0.3874

after 442010 years, the value is   0.0000
after 442011 years, the value is   0.0000
after 442012 years, the value is   0.0000
after 442013 years, the value is   0.0000
after 442014 years, the value is   0.0000
after 442015 years, the value is   0.0000
after 442016 years, the value is   0.0000
after 442017 years, the value is   0.0000
after 442018 years, the value is   0.0000
after 442019 years, the value is   0.0000
> >
```

Figure 4: The `R` console window showing an infinite loop. I clicked on `STOP` after 442019 trips through the loop.

A defensive programmer will protect against infinite loops. One way is to insert an arbitrary maximum number of trips through the loop. In the `cic()` function, it might be wise to stop if the target M isn't reached in 1000 years. Figure 5 has a version of the `cic()` function with these defensive features added. Lines 10 and 15 start `if` tests to check that $r > 0$ and $M > 1$. There are error messages if the conditions are violated. Lines 13 and 18 say to return the value 0. More importantly here, they say to stop executing commands in the function. This keeps the function from "operating" with bad argument values. Line 25 defines the maximum number of trips through the loop (more on this below). Line 38 is the `if` test to see whether you've reached that maximum number. If you have, print an error message and stop looping (set `continue` to `FALSE`).

It may seem simpler not to define the variable `Tmax` (line 25), and just make line 38 say: `if ( T > 1000 ){`. This would do the same thing, but it's bad programming practice because the constant is *hard wired*. The term "hard wired" comes from electrical wiring (or cabling, these days). Connections are hard wired if they are fastened in with bolts or something to make sure they connections are secure. A hard wired connection is hard to change if it needs to be changed. In programming, a "hard wired" number is put in so that it is hard to change. This isn't the case here. It's just as easy to change 1000 to 2000 in `Tmax = 1000` as in `if ( T > 1000){`. But often the parameter appears in more than one place. If you change it, you have to remember to change it everywhere.

It's very common to get that wrong. Also, line 25 with its comment makes it easier to see in reading the program that there is a maximum trip count and how to change it.

cicd.R

```
#  print compound interest until reaching a target
#  different way of running the while loop
#  with some defensive programming features added

cicd = function( r, M){

#    Argument checking: stop unless r > 0 and M > 1

     if ( r <= 0 ){
         output = sprintf("Function cicd got negative r =  %8.4f\n", r)
         cat(output)
         return(0)
       }
     if ( M <= 1 ){
         output = sprintf("Function cicd got M less than 1, M =  %8.4f\n", M)
         cat(output)
         return(0)
       }

     V = 1  # the value after some years, initialize to 1
     T = 0  # count the number of years

     continue = TRUE   #  controls the while loop, test at the end
     Tmax     = 1000   #  stop the loop after this many years even if V<M
     while ( continue) {
        output = sprintf("after %3d years, the value is %8.4f\n", T, V)
        cat(output)
        T = T + 1
        V = (1+r)*V

        if ( V > M ){         # check the termination criterion

            output = sprintf(" get a factor of %8.4f in %3d years\n", M, T)
            cat(output)
            continue = FALSE   # tell the loop to stop
          }
         if ( T > Tmax ){
            output = sprintf("Function cicd stopping for too many iterations\n")
            cat(output)
            continue = FALSE
          }

       }
     return(V)
}
```

Figure 5: The code from Figure 3 with defensive features.