ECON 442: Understanding R Code Professor Jack Rossbach

**1. Assignment**: Assignment is the process by which an expression (number, string, function, etc) is assigned to a variable.

Assignment can be done in R using either "="or the "<-".

Examples:

```
x=3
```

Sets $x$ equal to 3

```
y<-2x
```

computes 2*x (if x =3 then 2*x = 6) and assigns it to y

```
z = "hello"
```

Sets z equal to the string "hello" (note that smart quotes or angled quotes result in an error in R). You can alternatively use apostrophes, e.g. 'hello', but again not curved ones.

*Tip: Semi colons (;) can be used to separate lines of code as an alternative to enter/return/new line*

**2. Vectors**

Vectors are lists of vector objects (primarily numbers, logical statements, strings, and other vectors -- not functions). There are several ways of creating vectors, discussed below. After being created, the ith element of the vector can be accessed by putting "[i]" after the name of the vector. If you reference an element longer than the length of the vector, you will get a "NA" (not available/missing value); if you assign a value to an element of a vector longer than the current length, it

Ways to create vectors:

- The `vector()` function: used to tell R that a given variable will be used as a vector. Allows you to reference elements of the vector after being created (will result in an error if the variable is not first declared to be a vector).

  Example: typing `zz[2] = 4` will return an error if zz is not defined. Instead typing

  ```
  zz <- vector()
  zz[2] = 4
  ```

  will create the vector zz which will have length equal to two, the first element will be `NA` and the second element will be equal to 4.

- The `c()` function: used to create a vector from a set of given vector objects, each separated by a comma

  Examples: `my_vec = c(1,3,7)` will create a vector named `my_vec` with the first element equal to 1, the second equal to 3, and the third element equal to 7.

`test <- c(my_vec, "hello", "goodbye")` will return the vector

```
[1] "1"       "3"       "7"       "hello"   "goodbye"
```

Note that here the numbers have been changed into strings, since vectors can only handle one vector object type at a time.

*Tip: when you display a vector in the R console, there will be numbers in brackets on the left of the console. This is not part of your vector and is only displayed for your help. It displays the number of the element immediately to the right of the bracketed number. E.g. if it says [6] "hello" "goodbye" that means that the 6th element of the displayed vector is "hello".*

- The colon operator "a:b" is used to create a list from the first number, a, to the second number, b. It always increases or decreases by 1, depending which number is greater. If reaching b is not possible, it stops one step before reaching b.

  Examples: `1:5` returns `[1] 1 2 3 4 5.` `7:3` returns `[1] 7 6 5 4.`

  `1.5:3` returns `[1] 1.5 2.5`

**2. Operators**:

Tip for Operators: If both objects are vectors of the same length, everything is done element by element. If only one object is a vector of length greater than 1 and the other object is a scalar (vector of length 1) then the operation applies between the scalar and all elements of the vector.

**Arithmetic Operators** are used to perform arithmetic calculations. Addition (+), subtraction (-), multiplication (*), division (/), exponents or taking powers (^).

Examples: `c(2,3)+4` returns `[1] 6 7.` `c(10,20,30)/c(1,2,3)` returns `[1] 10 10 10.`

`c(2,4)^c(2,1/2)` returns `[1] 4 2`

**Relational Operators** are used to compare two expressions. Relational Operators can be can considered a test that returns TRUE if the relational operator holds and FALSE if it does not hold.

Here are the common relational operators (the R code in parenthesis): Equal to (==), Not Equal To (!=), less than and not equal to (<), less than or equal to (<=), greater than and not equal to (>), greater than or equal to (>=).

Examples: `2 != 3` returns `[1] TRUE`

`c(4,3,2) >= 3` returns `[1] TRUE TRUE FALSE`

`x=c("abc",4); "abc"==x` returns `[1] TRUE FALSE`

**Logical Operators** are used to combine TRUE and FALSE statements using and (&), or (|), or not(!).

Note that in computer programming **and** means that all statements must be TRUE, in contrast **or** means one or more statements must be TRUE (i.e. or **does not** mean that both are not true).

*Tip: For logical operations, TRUE == 1, False == 0*

Examples: `!TRUE` returns `FALSE` `TRUE&0` returns `FALSE`

`c(TRUE|TRUE&FALSE,!1)` returns `[1] TRUE FALSE`

**3. Conditional Statements or "If" Statements** are used to run a segment of code only under the condition that another expression is true. They work as follows

```
if(condition) {
      code to run
}
```

Having "{" and "}" brackets on separate lines and indenting the nested code are best practices for writing readable code, but do not alter how the code is executed in a plain if statement (as seen in the examples below). The code only runs if condition == TRUE.

Example: `if(TRUE) {print(1)}` returns `TRUE`

`if(0) {print("hello")}` does nothing since no code is executed

**"else if" Statements** are used equivalently to if statements, however they must immediately follow an if statement, and the condition is only tested if the first if statement did not run. For else if statements the **bracketing does matter** in terms of the else if statement being on the same line as the } from the previous if statement. The format for else if statement is

```
if(condition 1) {
      code to run if condition 1 is TRUE
} else if(other condition) {
      code to run if condition 1 is FALSE and condition 2 is TRUE
} else {
      code that runs if all previous conditions are FALSE
}
```

Example:

```
x <- 0.5
  if (x == 1){
    print('same')
  } else if (x > 1){
    print('bigger')
  } else {
    print('smaller')
  }
```

returns `"smaller"`

**4. Loops** can be used to repeatedly run a segment of code. Often times variables will change for each run.

**For Loops** work by defining a variable and giving a vector as an argument. The loop will execute the main code as many times as the length of the given vector. For the first run, it will set the variable equal to the first element of the vector, for the second run it will set the variable equal to the second element, and so on.

```
for(variable in vector) {
      run this code (setting variable equal to the ith element of
      the vector for the ith run)
}
```

Examples: `j=1; for(i in 1:3) {j=j*i};` will have j set equal to 6 after the loop is finished running (first run is 1*1, second run is 1*2, third run is 2*3)

```
for( var in c(1,-1,5,0)) {
      if(var<=0) {
            print(var)
      }
}
```

Will display `-1` and then `0`, each on its own line, and then not display any additional output.

**While Loops** work by defining a condition, and the loop will execute the main code as long as the condition is true. It is possible to get trapped in a while loop that will run forever, so it is important that the code you run each loop will eventually turn the condition FALSE. Ctrl+c will stop the code that is currently running to allow you to escape.

```
while(condition) {
      run this code while condition is TRUE


}
```

Examples: `j=1; while(j<=3) {j=j+1};` will return j equal to 4 (the third run sets j=4, after which the condition is FALSE).

```
myval = 100; test = 1; i=1;
while( test > .05) {
      test = myval^-i
      i = i+1
}
```

Will finish with `myval = 100`, `test = 0.01`, and `i=5`.

**5. Defined Functions** take objects as inputs (vector objects such as numbers or strings, but also objects such as other functions), and execute some code. What happens in the function stays local to the function (i.e. if you define a variable inside a function, it may not be available outside the function). The function will return whatever expression is computed in the last line before the end bracket "}", otherwise you can tell the function what it's output should be using the `return()` function.

You need to assign the function you define to a variable/object, so that you can later call the function with given inputs.

```
example_fun <- function(inputs) {
      code that runs

      return(what the function gives back as output)
}
```

Examples: `take_diff <- function(vec) {return(vec[1]-vec[2])};`

```
take_diff(c(3,1))
```

will return 2. If we run the above code first and then run the below

```
nested10 <- function(arg,fun) {
      fun(c(arg,10))
}
nested10(15,take_diff)
```

it will return 5 (it calls the function `take_diff`, and gives it a 2 element vector as an input. The first element of the vector is `arg` and the second element is `10`.

**6. Various Built in Functions You Should Know** (will expand)

- max()/min(): Return the largest/smallest value of all inputs (these are not optimizing functions/to be applied to functions to find the point that maximizes something)
- pmax()/pmin(): Given multiple vectors, returns the element by element largest/smallest value across the matrices
- length(): Returns how many elements are currently in a vector. Useful for loops if you want to do something for each element.
- Libraries are collections of functions. most libraries can be installed online through R directly by installing packages. After installing a package (only needs to be done once), you need to call the package using the library function. `library("package")` calls the library named package so you can use it's functions.
- We will mainly use the rootSolve package and the multiroot function (see previous R_programming notes)