# Getting started with R

These notes are for someone who can use a computer generally but has not done any programming or done other serious computer work. Everything below is here because I've helped a student do it. Just move quickly through the things you already know.

This is the first of several "handouts" explaining the `R` language. It describes the *command line* and the *environment*. It explains how *objects* are created and how their values change. It walks you through defining a *function* "at the command line". It ends with a bit of professionalism, which is carefully formatted output.

Working at the command line is convenient and quick if you're doing something very simple. But it is time consuming and frustrating for anything at all complicated. The next handout will take you away from the command line to files and code editors. These allow us to create and edit sequences of `R` commands without executing them. You can go back and fix mistakes before the whole thing is executed. You can put separate functions in separate files, which is a must for large codes.

As we go, there will be many points of professionalism and programming style. These are things we do to make the code easier to write, to *debug* (fix mistakes), to read, and to understand. Every applied mathematician today spends time computing, which means that she/he must do some coding. Good coding saves time in the long run, and usually in the short run. In this class, you will be expected to use good coding practices as they are presented. You will lose points on "perfectly good" working code if it doesn't doesn't conform to coding standards. Most coding and computing is teamwork. Your team partners will value your code more if it is easy for them to understand and modify.

## Get set up

Get the `R` app from (this should be a live link)

`https://www.r-project.org/` .

There is a good web site to help you get started here:

`http://scs.math.yorku.ca/index.php/R:_Getting_started_with_R` .

Follow the instructions there until you open on your screen the `R` app. On my imac laptop it looks like this (a picture of part of the screen):

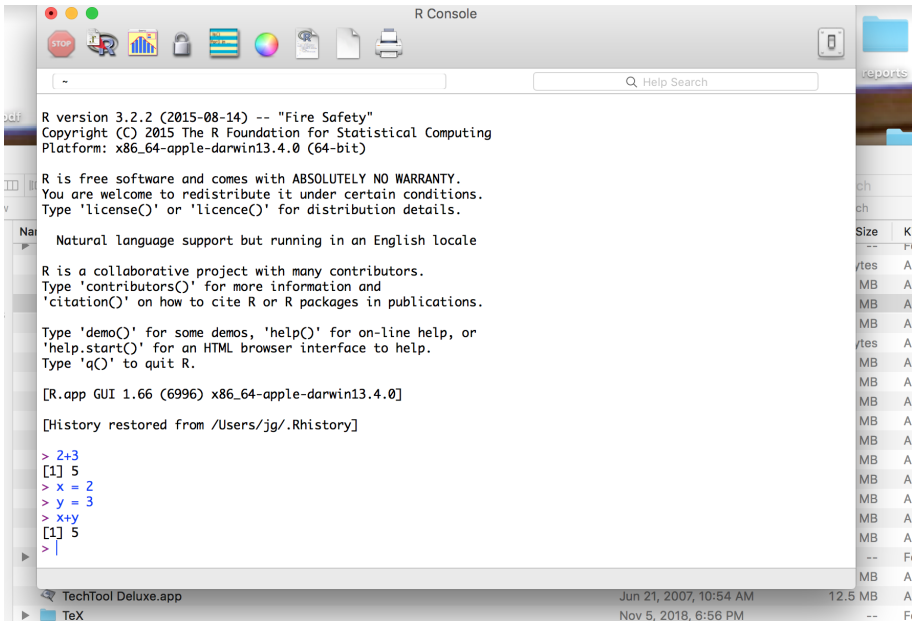


Figure 1: A screen capture of the R command window

The > on the bottom line of the R app window is a *command prompt*. What you type comes out there. You type and press **enter** and the R *interpreter* will read what you typed and do something. At the first prompt, I typed "2+3" then pressed **enter**, or `2+3 [enter]`. The R interpreter typed `[1] 5`. Don't worry about the `[1]` yet. The answer was 5. What I typed `2+3 [enter]`, was a *command*. What R did was to *interpret*, or *evaluate*, or *execute* the command (these words mean slightly different things).

## The R environment

When the R interpreter evaluates a command, it may type an answer if that's appropriate. It also might *update* the working *environment*. The environment is information that the R interpreter maintains and updates as it executes (evaluates) commands. When I gave the command `2+3 [enter]`, the interpreter evaluated the expression and typed the result, but it did not change the environment. When I gave the command `x=2 [enter]`, the interpreter didn't type anything back, except the next prompt. But it updated the environment. It added a *variable* (more properly called an *object*) called `x` and gave it the value 2. A command like this is an *assignment*. It assigns the variable `x` the value 2. My next command, which was the assignment `y=3 [enter]`, caused R to add `y` to the environment and give it the value 3. The next command, `x+y [enter]`, is an expression (to be evaluated), not an assignment. The evaluation uses information in the environment at that time. The environment has objects

called x and y with values 2 and 3. The expression gets the value 5.

```
[History restored from /Users/jg/.RI

> x = 2
> y = 3
> ls()
[1] "x" "y"
> x+y
[1] 5
> x = 4
> x+y
[1] 7
> x+z
Error: object 'z' not found
> z = x+y
> z
[1] 7
> x+z
[1] 11
>
```

Figure 2: Some commands that illustrate the R environment

Figure 2 illustrates the updating of the environment. The first two commands, `x=2 [enter]` and `y=3 [enter]`, are as before. The command `ls()` ("ls" is for "list", as in "list all ... ") prints a list of all objects currently in the environment. There are two, `x` and `y`. The command `x = 4 [enter]` changes the environment by changing the value the object `x`. This causes the command `x+y [enter]` to evaluate to 7, not 5. The command `x+z [enter]` is an expression that cannot be evaluated in the present environment because there is no variable (object) `z`. The R interpreter prints an *error message* saying what went wrong. The next command, `z = x+y [enter]` assigns object `z` the value that is the value of the expression `x+y` evaluated in the present environment. A new object called `z` is created by this command. The command `z [enter]` is an expression whose value is the value of `z`. It has the value we expect, 7. Finally, the command `x+z [enter]` returns the sum of the values of `x` and `z` in the present environment.

Many R programmers prefer to use `<-` rather than `=` for assignments. The command `x <- 2 [enter]` means the same thing as `x = 2 [enter]`. The arrow symbol makes it more clear what is happening – the value on the right is being put into the object on the left. But most other programming languages (C, C++, Java, Python, Matlab, FORTRAN, ...) use the `=` symbol for assignment. I use `=` because it comes naturally to me (FORTRAN programmer) and because it's easier for me to type. Feel free to use `<-` if you prefer.

## Types (classes)

Every object in R has a *type*, also called its *class* (these are not exactly the same thing). The objects in the examples above should (but don't, see below) all

have type `int` for *integer*. Integers can be positive or negative. An integer $n$ is represented exactly if $|n|$ is not too large. The operations $+$, $\times$ (denoted `*`) and $-$ are done exactly if the result is not too large. The rules of arithmetic are satisfied exactly. For example, `(x+y)+z` produces exactly the same result as `x+(y+z)` if all variables are integers.

*Floating point* is a type that represents general real numbers. It is called `double` (for *double precision*, or extra accurate) in `R`. It is rare that a real number is represented exactly in the computer, integers and some rational numbers being exceptions. The operations addition (`+`), multiplication (`*`), and division (`/`) are rarely done exactly. The rules of arithmetic may not be satisfied exactly. The term *floating point* refers to the decimal point, which "floats" in scientific notation. The number .000567 is written $5.67 \cdot 10^{-3}$ in scientific notation. `R` would type something like `5.67e-03`. The `e` is for *exponent*. I don't know why it gives the exponent as `-03` instead of just `-3`.

In `R`, a number is automatically given the data type `double` even if it's an integer. The function `as.integer([value])` forces the given value to be represented as an integer, not a `double`. You might do this if you want arithmetic operations to be exact. There is more on *functions* coming.

Data can be non-numerical, such as a name or a piece of text. The *character string* (often called *string*, but called *character* in `R`) data type is for this. A character string is a sequence (a *string*) of characters. A *character* is a letter, a number, a punctuation mark, etc. You represent a string in `R` by putting the characters in quotation marks. For example the assignment `x = "Hello world" [enter]` creates (or re-purposes if an object called `x` already exists in the environment) an object called `x` with type `character` and value "Hello world".

Anything `R` prints is a string. Working with strings is *string manipulation*. We do string manipulation to process character data, and to make output easier to read. The `R` function `paste` makes a new string by pasting (technically, *concatenating*) the strings given as *arguments* to the `paste` function. (There is more on functions and arguments below.)

Figure 3 illustrates these three `R` types. The assignment command `x = 2 [enter]` creates an object called `x` and gives it the numeric value `2`. The `typeof()` function in `R` tells you (technically, *returns*) a string that represents the data type of the value of the object. In this case, `R` has represented the number `2` as a double precision floating point number, or `double`. The `R` function `as.integer()` returns a value that is an integer, so `typeof(x)` now returns `integer` instead of `double`. The environment has changed. The old `x` is gone (repurposed) and the current `x` is an integer. The square root function in `R` is `sqrt()`. The command `y = sqrt(x)` *calls* the `sqrt()` function with *argument* `x` and *returns* a double precision floating point number that best approximates $\sqrt{x}$. This value is assigned to the object `y`, which gets type `double`. But `y` is not exactly $\sqrt{x}$ and multiplication is not done exactly. That's why `x - (y*y)` (an `R` implementation of $x - y^2$) doesn't evaluate to zero. But notice that the error (called *roundoff error*) is small, of size $5 \cdot 10^{-16}$. The next command `x = "Hello"` reassigns (repurposes) the object `x` to the character string "Hello".

In the present environment, `typeof(x)` is not numeric, but `character`. The old value $x = 2$ is gone from the environment. The `R` function `paste` returns a string that is the concatenation of the strings given as arguments. In this case the arguments are `x` and `y`, which have values `"Hello` and `"world"`. This return value is assigned to the object called `z`. In the present environment, `z` has value `"Hello world"`.

```
[R.app GUI 1.66 (6996) x86_64-apple-darwin13.4.0]

[History restored from /Users/jg/.Rhistory]

> x = 2
> typeof(x)
[1] "double"
> x = as.integer(2)
> typeof(x)
[1] "integer"
> y = sqrt(x)
> typeof(y)
[1] "double"
> x - (y*y)
[1] -4.440892e-16
> x = "Hello"
> typeof(x)
[1] "character"
> y = "world"
> z = paste( x, y)
> z
[1] "Hello world"
>
```

Figure 3: Numeric and string types in `R`

## Functions

*Coding* usually means creating a sequence of commands that does something specific. In `R`, this is often done by defining a *function*. In `R`, a function has a sequence of commands, some *arguments*, and an environment. The creates an object, which is its *return value*. You communicate to a function by giving the arguments and creating an environment. The function does some stuff and communicates back to you by *passing* its return value. A function becomes part of the environment when you *define* it. After that, you can *call*, or *evoke* the function using its name.

Figure 4 shows how functions work in `R`. The first two assignment commands put objects with type `string` and names `x` and `f` into the environment. The next command starts the definition of a function. It creates a function object with name `F` and type `closure` (no idea how they picked this name for the type of function objects). After `=`, the word `function` is a *keyword* that tells `R` that it's a function being defined. After the keyword `function` comes parentheses (parens) that contain a sequence of *arguments*. There can be any number of arguments, 0, 1, 2, ... . This function has one argument, called `x`. After `(x)` is {, which is called an *open curley*. Curley is for *curley brace*, the name of the symbols { and }. This open curley opens a *code block*, which is the sequence of commands that defines the function. When I typed `F = function(x){ [enter]`, the `R` interpreter did not give me another command prompt (the `>` symbol). Instead

I got the *continuation prompt* symbol `+`. This is because the command that started with `F = function(···` is not finished until it gets a closed curley } to match the open curley on this line. It's called *continuation* when it takes more than one line to type a command. Every line after the first is a *continuation line*.

The first command of the function definition is `y = x*x`. When the function is executed, it will have an environment in which `x` has the value that was *passed* to it when the function was called (see below). This command creates object `y` in the environment of the function. It assigns `y` the value `x*x`. This would be an error if `x` had the value `"Hello"`, but when the function is evoked, the `x` from the outside environment (if there is one) is replaced in the function's environment with the *calling argument value* (see below). The second command in the function definition is `return(y-2) [enter]`. This will cause the value `y-2` to be *passed back* as the *return value* of the function. The `return()` command tells the `R` interpreter to stop executing commands from the function and go back to taking them from the command line. The next line just has a *close curley*. This tells the interpreter than you are done defining the function. It marks the end of the command that started with `F = function(···`. That function definition command is now executed. Executing the function definition command adds an object `F` to the environment and stores the definition of `F` (the commands between the curleys) as the "value" of `F`. That happens without returning anything, so the line } `[enter]` just led to the next command prompt `>`.

I typed the command `f(3) [enter]`. The parens after `f` tell the interpreter that `f` is a function. But `f` isn't a function. It's the string `"goodbye`. The `R` interpreter is *case sensitive*, which means that capital letters `F` are considered different from lower case letters `f`. The next `ls() [enter]` command returns a list of all the objects in the environment. There are three, the string `f`, the function `F`, and the string `x`.

The next command, `F(3)` *evokes* the function `F`. The argument `3` in parens is *passed* to the function. The `R` interpreter creates a new temporary environment for `F` and creates an object `x` with value `3`. The old value `"Hello"` from the outside is *masked* by this temporary new value. It then executes the commands in the `F` definition, starting with `y = x*x [enter]`. That creates an object `y` in the `F` environment and assigns it the value `x*x`, which is `9`. Next, it executes the command `return(y-2) [enter]`. This returns the value `y-2` (which evaluates to `7` in the `F` environment) to the outside environment. Then it "forgets" (technically, *garbage collects*) the `F` environment it created. The return value is printed. That's the `7` on the next line. With the `F` environment gone, the object `x` reverts to its original value, `"Hello"`. The object `y` from the `F` environment is gone. The object `F` is still there. It has type `closure`.

```
[R.app GUI 1.66 (6996) x86_64-apple-darw

[History restored from /Users/jg/.Rhist

> x = "Hello"
> f = "goodbye"
> F = function(x){
+     y = x*x
+     return(y-2)
+   }
> f(3)
Error: could not find function "f"
> ls()
[1] "f" "F" "x"
> F(3)
[1] 7
> x
[1] "Hello"
> y
Error: object 'y' not found
> typeof(F)
[1] "closure"
>
```

Figure 4: Define and use a function in R

## Formatting

In this context, *formatting* means creating a character string that represents the value of some other type of object. R has built-in formatting conventions for numerical values, which you can see at work in Figure 3. R also has built-in formatting functions that let you control formatting. You will use this control in every assignment, to make the output clear and easy to read.

The R function `sprintf()` is one way to format numbers into strings. It looks clumsy at first, but you will see that it is a simple way to do what it does. The `sprintf` and function are adapted from the C programming language. It started with a command called `print`, which became `printf` because it was printing to a file, which became `sprintf` when it became a function that returns the string rather than sending it to a file. The first argument to the function `sprintf` is the *format string*, which specifies the format you want to get. The rest of the arguments are the numbers (or, later, other things) that you want to format.

Figure 5 illustrates the mechanics of `sprintf()`. The first two assignment commands illustrate that object names can be more than one letter. It can be good programming practice (professionalism) to choose the name to make it clear to the reader what the object value represents. The command that assigns `string1` uses `sprintf()` with only a formatting string. You would get the same `string1` with the simpler assignment `string1 = "Hello world"`. The command `cat(string1) [enter]` uses the R built-in function `cat()`. This also comes from C, where it was short for *catenate*, which is how computer people say *concatenate*. It tells R to type out the argument without typing `[1]` first. It's a professional touch you always should use. The `string2` assignment formats the number `allowance` into the output string. The percent sign `%` in the format

string tells `R`: put a formatted number here. The `4` after the `%` says: use 4 characters for the number. The `2` after the `.` says: put two digits after the decimal point. The `f` after the `2` says: use *fixed point* format, and I'm done formatting this number. This creates a string of four characters, which is `3.50`. The rest of the characters in the output string come from the formatting string. The dollar sign `$` is just another character.

`Fixed point` means that the decimal point goes after the "one's digit" (which is `3` here). This is convenient for formatting numbers that are not too big or too small. The *exponential* format `e` is better for very big or very small numbers. The `string3` assignment uses `e` format for a number that is easier to understand in exponential format. It asks for `8` characters in all, of which `2` come after the decimal point. The last four characters are `e+06`, which means $\cdot 10^6$. The first four characters are the *mantissa*, which is `4.37`. The output `4.37e+06` means $4.37 \cdot 10^6$. Note that the actual number `4365...` has been (correctly) rounded up, not *truncated*. Truncation, just leaving off digits, would have given `4.36e+06`. The `string4` assignment asks for `10` characters in all instead of `8` and `3` digits after the decimal point rather than `2`. This leaves one extra, unused, character, which becomes a *leading blank*. You can see that `string4` has a blank between `$` and `4.365`.

You will often need to fiddle with the format numbers to get the numbers just as you want them to be. It's frustrating to do this at the command line, as it is done here. Next week we will do it using an editor, which is "the right way" to do it.

The `string5` assignment illustrates that you can format more than one number in a single output string. Look carefully and you can see the blanks before `4.60` and `1.10` that come from asking for 5 characters instead of 4. `R` gave us `4.60` rather than `4.6` because we asked for two digits after the decimal point. We get them even if they are zero.

```
[History restored from /Users/jg/.Rhistory]

> allowance = 3.50
> budget = 4365489
> string1 = sprintf("Hello world")
> string1
[1] "Hello world"
> cat(string1)
Hello world
> string2 = sprintf("Hello kid, you have $%4.2f this week", allowance)
> cat(string2)
Hello kid, you have $3.50 this week
> string3 = sprintf("Hello Senator, you have $%8.2e this week", budget)
> cat(string3)
Hello Senator, you have $4.37e+06 this week
> string4 = sprintf("Hello Senator, you have $%10.3e this week", budget)
> cat(string4)
Hello Senator, you have $ 4.365e+06 this week
> x = -3.5
> y = 4.6
> z = x+y
> string5 = sprintf("%5.2f plus %5.2f is %5.2f", x, y, z)
> cat(string5)
-3.50 plus  4.60 is  1.10
>
```

Figure 5: Using `sprintf()` to format numbers for printing

## Calculations in R

You already have seen that R has many built-in functions. They are part of the environment even though they don't show up in the `ls()` object list. The function `exp(x)` calculates the exponential $e^x$. For example, `exp(1) [enter]` returns `[1] 2.718282`. The function `log(x)` returns $\ln(x)$, the log base $e$. For example, `log(2.718) [enter]` returns `0.9998963`. This seems right since 2.718 is a little less than $e$, so $\ln(2.718)$ should be a little less than $\ln(e) = 1$.

Elementary arithmetic expressions in R use the operation symbols (`+, *, -, /`) and parens. Exponentiation is represented with `**`. For example, this expression

$$\left(1 + \frac{r}{m}\right)^{mt}$$

could be expressed as

```
(1 + (r/m))**(m*t) .
```

Be careful with parentheses. The expression `(1 + (r/m))**m*t` is interpreted as

$$\left[\left(1 + \frac{r}{m}\right)^{m}\right] t \, .$$

R has *precedence rules* that determine the order of arithmetic operations. Operations with higher precedence are done first, using the arguments next to them. The exponentiation operation has the highest precedence so it is done first. That means that `a*b**c` does $b^c$ first then multiplies by $a$, which gives $a\,(b^c)$. If you want $(ab)^c$, you must use parentheses, as `(a*b)**c`.

It is good programming practice to use parentheses even if you know the precedence rules will do what you want without them. For example, if you want $a\,(b^c)$, it's good to write `a*(b**c)`, even though the parens are unnecessary. This will help the person reading your code, who may not understand the precedence rules as well as you do. More importantly, it can save you from the programming mistake (bug) of getting the parens wrong. A bug like that can be hard to find because the expression looks right – like the mathematical formula. In the expression above `(1+(r/m)**(m*t)`, the parens around `r/m` are unnecessary because division has higher precedence than addition. I put them in to make the expression clearer.

Short variable names can make formulas easier to read. Suppose you had used `interest_rate` instead of `r`, and `number_of_compoundings` instead of `m`, and `loan_duration` instead of `t`. The expression would be

```
(1 + (interest_rate/number_of_compoundings))**(number_of_compoundings*loan_duration) .
```

Modern programming advice suggests that you should not use very long lines like this. Pure programming classes may advise you to use longer variable names that are more descriptive. There is value in that too, as you don't have to remember what a variable like `m` represents. You can use *comments* (described in the next handout) to make programs with short variable names easy to read. It is a tradeoff in the end. Excellent professional developers argue over things like this. Just being aware of the issue will make you a better developer.