Chapter 2

The Very Basics of the R Interpreter

OK, the computer is fired up. We have **R** installed. It is time to get started.

1. Start **R** by double-clicking on the **R** desktop icon.
2. Alternatively, open the terminal/console, and navigate to the directory in which **R** is installed: For example, go to `C:\Programs\R-2.15.2\bin` and type `R.exe` and hit enter.[1]

2.1 R as a Calculator

Depending on how you started **R**, you should see either the **R**-console or an **R** session should have spawned at the command line. Either way, you should see something like this:

```
R version 2.15.2 (2012-10-26) -- "Trick or Treat"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

[1]If **R** is on your system's `PATH` you can skip the navigation step.

Note the > at the bottom. Whenever you see this symbol, it means that **R** is not doing anything and just waiting for your input. It's called the prompt. Let's break the rule about text editors and type directly into the console.

2.1.1 Basic Arithmetic

```
> 1 + 2
[1] 3
```

When **R** is running it stores all variables, data, functions, and results in the active memory of your computer. In the example above, **R** knows 1 and 2, as well as the basic *operator* "+". It now creates an *object* in the active memory containing the results of the computation and implicitly executes a function to print the content of the results *object* to the screen. **R** let's you know that what it is displaying in the square brackets [ ]. In our case this *object* contains only one value and **R** tells you that it has printed the first and only value of our results.

You can scroll through previous commands you've entered by using the up (↑) and down (↓) keys on your keyboard. Often instead of giving you the prompt again, **R** will instead display a "+" (aka the continuation line). This means you entered an incomplete command and that **R** is waiting for more input. (Note **R** is not reporting the [ ], as it is not displaying any content from an object stored in memory.)

```
> 1 +
+
```

If you are not sure what's going on, and why **R** is asking for additional input, or what input it wants, you can just hit `Esc` or `Ctrl-C`. It will tell **R** to forget it and bring back the prompt. Of course we know that **R** wants another summand in this case. So let's give **R** what it wants.

```
> 1 +
+ 2
[1] 3
```

Same problem here. **R** is waiting for you to close the bracket.

```
> 7 / (1 + 3
+ )
[1] 1.75
```

R understands the following basic operators:

1. $+$ and $-$ for addition and subtraction
2. $*$ and / for multiplication and division
3. $\wedge$ for exponents
4. %% is the modulo operator
5. %\% for integer division

R observes standard rules of operator precedence but you can use brakets if you don't remember those lessons your elementary school math teacher tried to hammer home.

This,

```
> 7 / (1 + 3)
[1] 1.75
```

is not the same as this:

```
> 7 / 1 + 3
[1] 10
```

2.1.2 Comments and Spacing

Comments

Another important operator is the comment operator #. Whenever R encounters this operator, it will ignore everything printed after it (in the current line). As can be inferred from its name, this is extremely useful to annotate your code.[2]

```
> 1 + 2 + 3 # Here R does some serious Maths.
[1] 6
```

[2]You can never use this functionality enough. Always annotate your code. Say you are writing a program to implement some non-standard estimator. The code works. Your results are nice and you submit them as a paper. You wait six months for the reviewers of your favorite journal to get back to you. When they do they mention that you should compute robust standard errors. You open up the code for your program but don't remember what you did and what any of it means. If you had annotated each line or section, you would probably be able to make sense of it all. Long story short. Annotate, annotate, annotate.

Be careful.

```
> 1 + 2 # + 20
[1] 3
```

Misplaced comments can break your code.

```
> 1 + # 2
+
```

Spacing

For the most part, R does not care about spacing. This:

```
> 1 + 2
[1] 3
```

produces the same result as this:

```
> 1     +          2
[1] 3
```

Spaces of course matter when you are dealing with character strings. This:

```
> print("Strings obey spacing.")
[1] "Strings obey spacing."
```

is clearly a different string than this:

```
> print(" Strings   obey       spacing  .   ")
[1] " Strings   obey       spacing  .   "
```

Semi-Colons

There is another special character, the semi-colon. The semi-colon is an important part of R's control structure. We have seen that R evaluates code line by line. A linebreak tells R that a statement is to be evaluated. Instead of a linebreak, you can use a semicolon to tell R where statements end.

```
> print("HELLO")
[1] "HELLO"

> print("HELLO") print("WORLD")
Error: unexpected symbol in "print("HELLO") print"

> print("HELLO"); print("WORLD")
[1] "HELLO"
[1] "WORLD"
```

2.1.3 Basic Functions

R comes with a slew of pre-installed functions. These functions are installed as part of the `base` package which is located in your `\library` directory. R treats all functions like *objects*. All functions have names and take arguments in parentheses: `function(...)`.

Let's consider the `print()` function. This function simply asks R to print *objects* to the screen. We can ask R to print known *objects*.

```
> print(1)
[1] 1
```

We can tell R to print any object, including a character string. Character strings are enclosed by quotation marks.

```
> print("We need more coffee!")
[1] "We need more coffee!"
```

Importantly, we can ask R to print named functions. Consider the function `exp()`. The function `exp()` computes the exponential function. Let's ask R to print it by using the function name as the argument for `print()`.

```
> print(exp)
function (x)  .Primitive("exp")
```

What **R** is telling you here is that `exp()` is a known function which takes one arguments `x`. (It also tells you what type or class of *object* `exp` is. The type here is .Primitive, which is a basic **R** function.) Let's try it.

```
> exp(x = 1)
[1] 2.718282
```

Let's ask **R** to print another function, say `log()` which computes logarithms. Here **R** reports that the function takes two arguments, separated by a comma. It needs and `x` and you can specify the `base`. If you do not specify the second argument, **R** will default to `base = exp(1)` (i.e. the natural logarithm).

```
> print(log)
function (x, base = exp(1))  .Primitive("log")
```

Ok ... let's try.

```
> log(x = 10, base = exp(1))
[1] 2.302585
```

Or, a log with base 10 . . .

```
> log(x = 10, base = 10)
[1] 1
```

R is pretty smart. You do not need to tell **R** all the specifics. Whenever you use a function, **R** knows what arguments can be supplied to the function. So you can simplify things, like this:

```
> log(10)
[1] 2.302585
```

R knows that the first argument you supply is the `x` and since you did not add anything else to the function, it returns the default logarithm (i.e. `base = exp(1)`)

You can do the same for the second argument. **R** knows that the second argument has to be for the `base`.

```
> log(10, 10)
[1] 1
```

Below is an excerpt of some of the basic mathematical functions R knows.

- `print()` – prints objects
- `log()` – computes logarithms
- `exp()` – computes the exponential function
- `sqrt()` – takes the square root
- `abs()` – returns the absolute value
- `sin()` – returns the sine
- `cos()` – returns the cosine
- `tan()` – returns the tangent
- `asin()` – returns the arc-sine
- `factorial()` – returns the factorial
- `sign()` – returns the sign (negative or positive)
- `round()` – rounds the input to the desired digit
- etc, etc, ...

2.1.4 Logical Operators

Among the most used features of R are logical operators. You will use these throughout your code and they are crucial for all sorts of data manipulation. When R evaluates statements containing logical operators it will return either `TRUE` or `FALSE`. Below is a list of most of them.

1.	`<`	less than
2.	`<=`	less than or equal to
3.	`>`	greater than
4.	`>=`	greater than or equal to
5.	`==`	equal
6.	`!=`	not equal
7.	`&`	and
8.	`|`	or

Let's try them out:

```
> 1 == 1
[1] TRUE

> 1 == 2
[1] FALSE

> 1 != 2
[1] TRUE

> 1 <= 2 & 1 <= 3
[1] TRUE

> 1 == 1 | 1 == 2
[1] TRUE

> 1 > 1 | 1 > 2 & 3 == 3
[1] FALSE

> 1 > 1 & 1 > 2 & 1 > 3
[1] FALSE
```

2.1.5 R's Help Function

At this point it should be obvious how to use R as a calculator. In this section, I will outline how you can ask R for help. The first thing to do if you have a question about one of R's functions is to ask R. This is important and you will use this functionality a lot.

Known Functions

Recall our example of computing a base 10 logarithm by calling `log()` with the argument `base = 10`? Assume we didn't know that `log()` had an argument called `base`, how could we have found out? We can simply pull up a help page for the `log()` function, like this:

```
> ?log
>
```

A help page will appear with the following sections (there may be others):

→ **Description**: purpose of the function

→ **Usage**: an example of a typical implementation

→ **Arguments**: a list of the arguments you can supply to the function and what each does

→ **Details**: more detailed information about the function and its arguments

→ **Value**: information about the likely output of a function (e.g. does the function return an integer, or a list, or a matrix, or something different)

→ **See Also**: a list of useful related functions

→ **References**: citations which can often be very useful

→ **Examples**: example code

Using the ? command is generally a good start. And more likely than not, it will provide you with the answer you need. In many instances, however, the help is not helpful and it appears it was written in code. R programmers are not always good at explaining things in plain English. A good example is the help page for the straightforward function `sort()`. It is incomprehensible. In such cases, it may be useful to use the `example()` function, which displays the examples from the help page. These may or may not be helpful.

For some functions, especially basic operators, ? may not work. In those cases you can use the `help()` function:

```
> help("+")
>
```

Unknown Functions

It maybe the case that you know what you want to do but you don't know how to do it in R. Say you want to estimate a logit but don't know how. Instead of using the ?, you can use two ??. This will initiate the help search page, and search the documentation for packages you have installed for that specific keyword. (This is the same as: `help.search("logit")`.)

```
> ??logit
>
```

An alternative is to use the `appropos()` function. It will return a list of all functions known to R containing the search term (e.g., "mean").

```
> apropos(mean)
 [1] ".colMeans"  ".rowMeans"     "colMeans"   "kmeans"  "mean"
 [2] "rowMeans"   "weighted.mean" "mean.Date"
```

You can now use ? to find out more.

```
> ?rowMeans
>
```