Introduction to R

Alternative to our doc: An Introduction to R by CRAN

What is R and why R?

R (programming language)

From Wikipedia, the free encyclopedia

R is a programming language and free software environment for statistical computing and graphics supported by the R Core Team and the R Foundation for Statistical Computing.[7] The R language is widely used among statisticians and data miners for developing statistical software and data analysis. Polls, data mining surveys, and studies of scholarly literature databases show substantial increases in R's popularity;[8] since August 2021, R ranks 14th in the TIOBE index, a measure of popularity of programming languages.[9]

As you will see in the course, there are a lot of calculations and computations that are too difficult, too annoying, or outright impossible to do by hand.
Since the Year of Our Lord 2021, we can delegate the boring stuff to computers.

Q: How familiar/comfortable are you with programming?

Fx:

> The sum of the primes below 10 is 2 + 3 + 5 + 7 = 17.
> Find the sum of all the primes below two million.

1. I know *exactly* how to do this
2. I know how to start but I can figure it out
3. I'm not sure but give me some time and I'll manage
4. I have no clue

Installation:

Follow installation instructions on the R website. We will also use the RStudio IDE.
Some minutes for troubleshooting.

Getting started

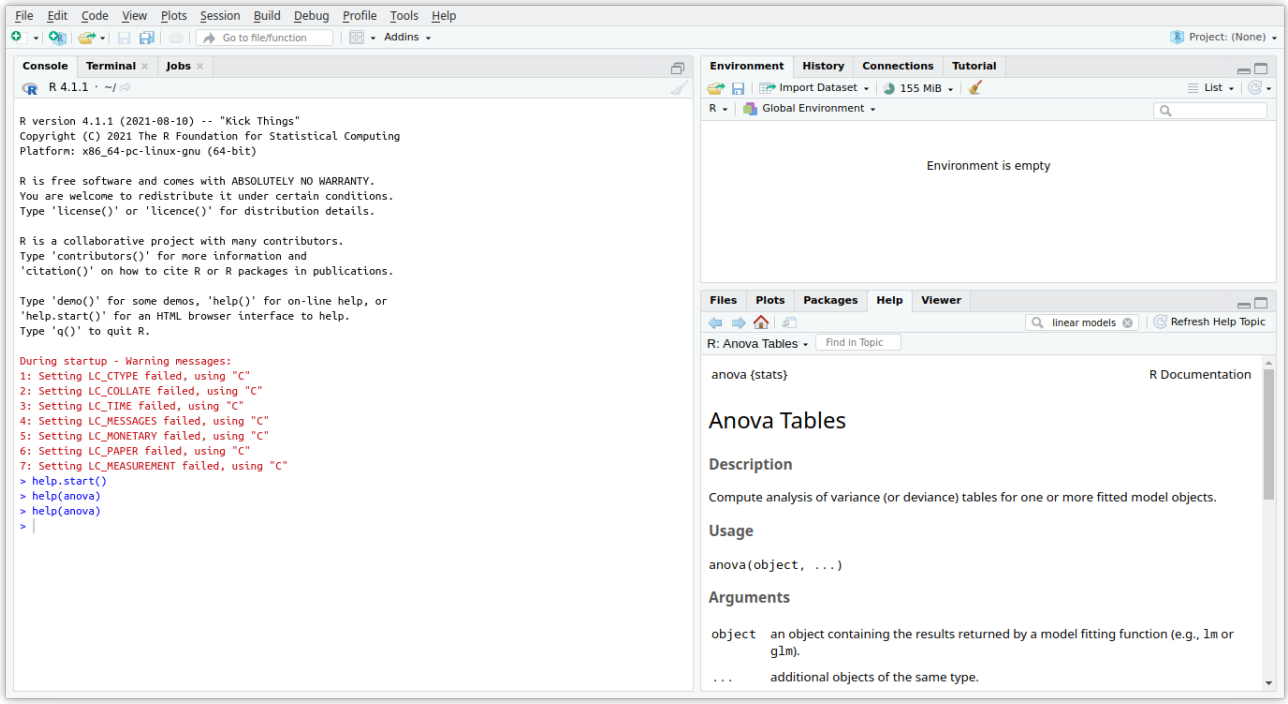
There is a command prompt

```
>
```

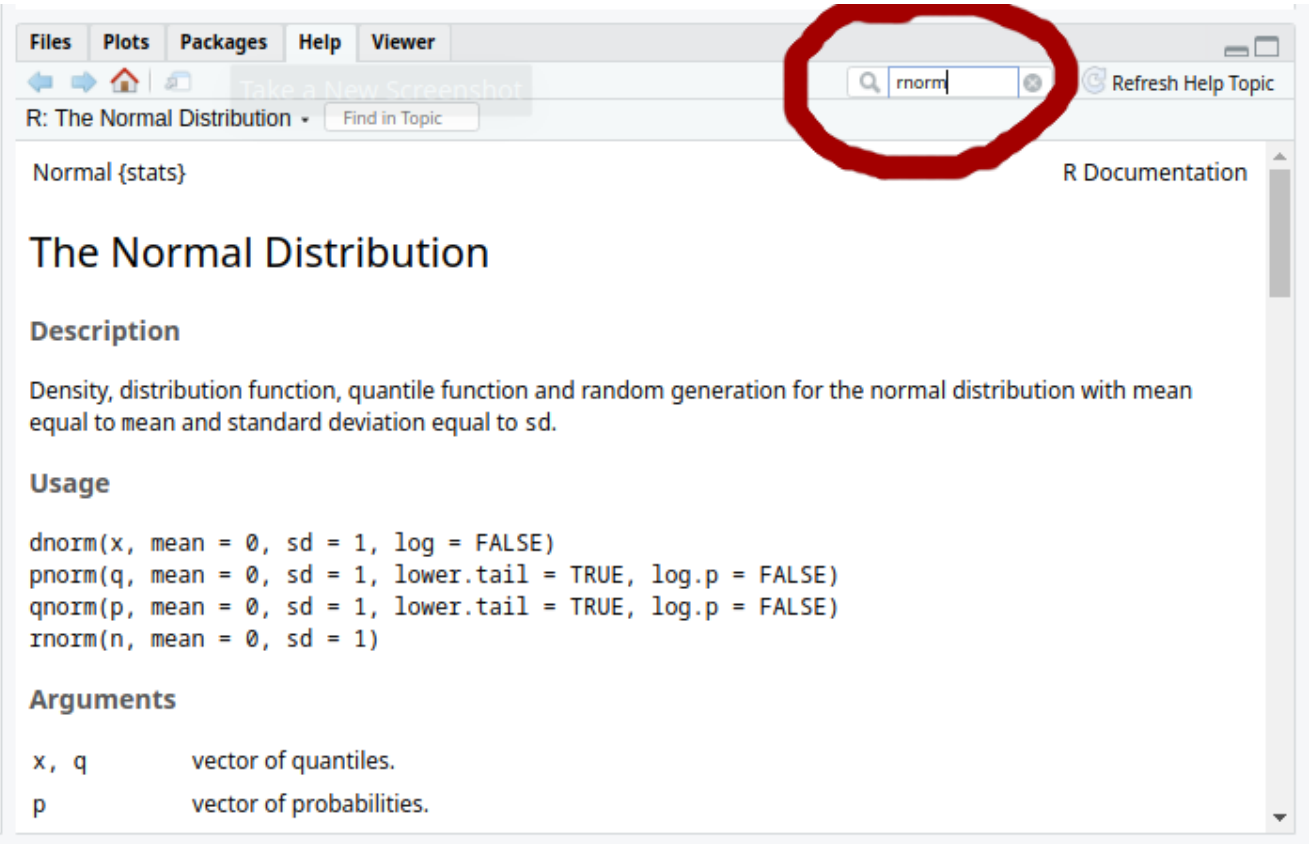
Here we type commands. We can ask for help:

```
help(anova)
```

If using RStudio, this opens the docpage in the window to the right.



In RStudio, we can also use the searchbar in the Help tab on the right:



We will talk about how to read this documentation below.

Basic syntax

Statements end with `;`, comments can be added with `#`.

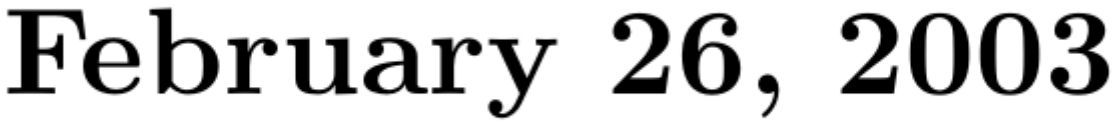
Addition, multiplication, substraction, division as per usual. Raising to some power is done with `a^b` or `a**b`, modulo function is `%%`.

Assignment of variables with `=`, `->` or `<-`:

```
> x <- 2
> 4 -> y
> x+y
[1] 6
```

Why?
Who hurt you, R

Note: The document states that we can use `_` to assign variables but I tried and didn't work (both RStudio and command-line). This was probably deprecated sometime between now and...



Oh.
Indeed, there is no mention of assignment of variables with `_` in the official Intro to R tutorial from CRAN.

Boolean values are `FALSE` and `TRUE` (also `0` for true and any non-zero for false).

You can list all the variables you have stored with `ls()`:

```
> ls()
[1] "x" "y"
```

And you can delete it with `rm`.

Arrays (vectors)

You can construct vectors with the `c` function:

```
> vec1 <- c(1,2,3,4)
> vec1
[1] 1 2 3 4
```

You can create arrays of consecutive integers with `min:max`:

```
> 5:10
[1] 5 6 7 8 9 10
```

Note that both limits are *inclusive*.
We can also create sequences with `seq`. The default syntax is

```
seq(min, max, step)
```

But we can pass the keyword argument `length` to set the length of the array instead of the step:

```
> seq(5,10,0.5) #same as seq(5,10,by=0.5)
[1]  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
> seq(5,10,length=8)
[1]  5.000000  5.714286  6.428571  7.142857  7.857143  8.571429  9.285714
[8] 10.000000
```

Note that the limits are *inclusive*.

Matrices:

Constructed from the unrolled version + number of rows or columns:

```
> matEntries <- 1:16
> matrix(matEntries,ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> matrix(matEntries,ncol=2)
     [,1] [,2]
[1,]    1    9
[2,]    2   10
```

```
[3,]    3   11
[4,]    4   12
[5,]    5   13
[6,]    6   14
[7,]    7   15
[8,]    8   16
> matrix(matEntries,ncol=8)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    3    5    7    9   11   13   15
[2,]    2    4    6    8   10   12   14   16
```

Note that the entries get rolled up column by column.

Entry-wise product is with `*`, and matrix multiplication is with `%*%`. Matrices can be inverted with `solve`.

Reading docstrings and function parameters

As we saw above, we can look for the documentation of different things with either the `help` command or using the searchbar in the Help window. For example, if I type `help(runif)` I get the following (massive) docstring

> # The Uniform Distribution
>
> ## Description
>
> These functions provide information about the uniform distribution on the interval from min to max. dunif gives the density, punif gives the distribution function qunif gives the quantile function and runif generates random deviates.
>
> ## Usage
>
> ```
> dunif(x, min = 0, max = 1, log = FALSE)
> punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
> qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
> runif(n, min = 0, max = 1)
> ```
>
> ## Arguments
>
> `x, q` vector of quantiles.
> `p` vector of probabilities.
> `n` number of observations. If length(n) > 1, the length is taken to be the number required.

`min, max` lower and upper limits of the distribution. Must be finite.
`log, log.p` logical; if TRUE, probabilities p are given as log(p).
`lower.tail` logical; if TRUE (default), probabilities are P[X ≤ x], otherwise, P[X > x].

Details

If min or max are not specified they assume the default values of 0 and 1 respectively.
The uniform distribution has density
f(x) = 1/(max-min)
for min ≤ x ≤ max.
For the case of u := min == max, the limit case of X == u is assumed, although there is no density in that case and dunif will return NaN (the error condition).
runif will not generate either of the extreme values unless max = min or max-min is small compared to min, and in particular not for the default arguments.

Value

dunif gives the density, punif gives the distribution function, qunif gives the quantile function, and runif generates random deviates.
The length of the result is determined by n for runif, and is the maximum of the lengths of the numerical arguments for the other functions.
The numerical arguments other than n are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The characteristics of output from pseudo-random number generators (such as precision and periodicity) vary widely. See .Random.seed for more information on R's random number generation algorithms.

References

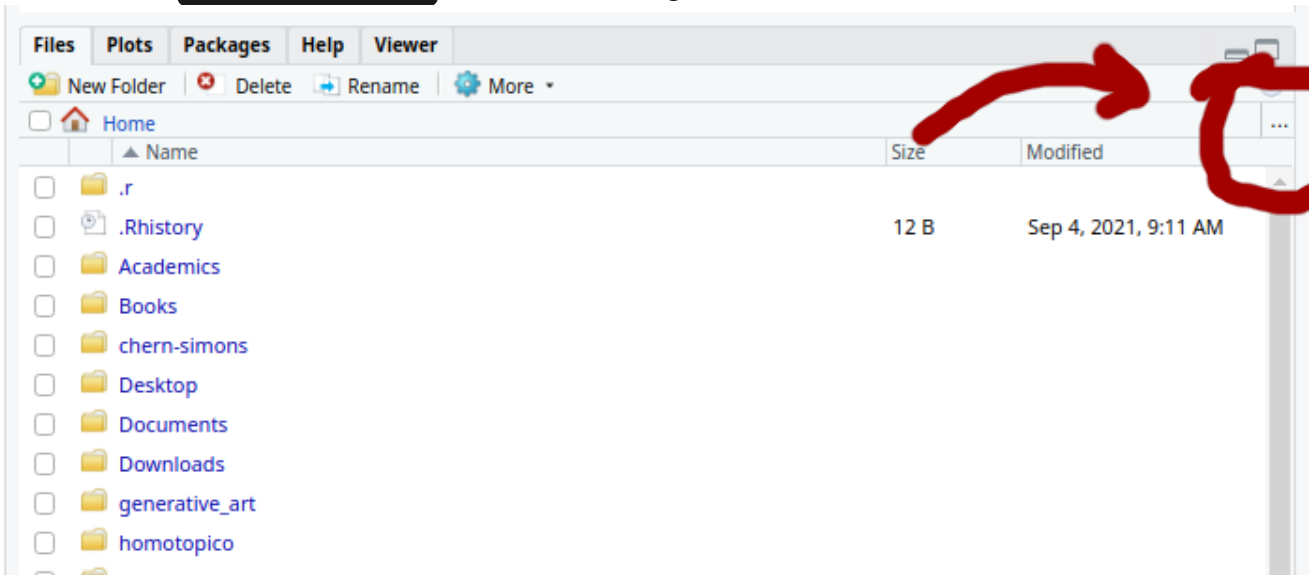
Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language. Wadsworth & Brooks/Cole.
See Also
RNG about random number generation in R.
Distributions for other standard distributions.
Examples
u ← runif(20)
/## The following relations always hold :

> punif(u) == u
> dunif(u) == 1
> var(runif(10000)) #- ~ = 1/12 = .08333

Let's focus on `runif(n, min = 0, max = 1)`. If we want to use it to generate 10 uniformly distributed numbers between 0 and 1, we must do

```
> runif(10)
```

Note that the docstring showed that `runif` has 3 parameters `n, min, max`, but in the call above we only used one. This is because `min` and `max` have *default values* of 0 and 1, respectively. We can identify when a parameter has a default value when it is defined like `param=something` in the docstring.



Exercises to try:

Most exercises are just about syntax (so solved by looking at the documentation or by googling). These are a bit more fun:

7. Using the functions `runif()` and `trunc()` simulate 20 die throws.
8. Generate a randomisation list of size 150 for three treatment groups “Placebo”, “Drug A” and “Drug B”.

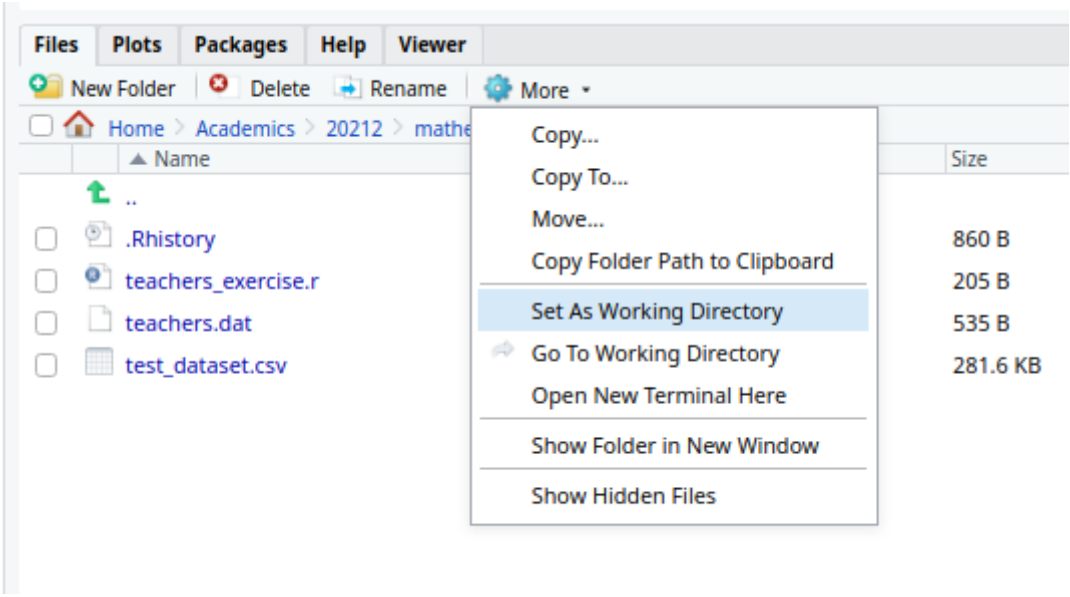
Loading data and dataframes

Before we even try to load data in R and RStudio, we have to let it know what the working directory is. The **working directory** is the place where R will store files and where it will look for data.

In barebones R (i.e. the command console) we can set the working directory with

```
> setwd("path/to/directory")
```

However, in RStudio we can set it manually. In the lower right pane, click on the **Files** tab. Here you can either navigate directly to the folder you want, or use the `...` icon on the right and select your preferred directory. After you've done that, click on the Gear icon and then on **Set as Working Directory**.



Once you do this, you've told R *where* to look for files.

Think of a dataframe like an Excel spreadsheet.

We load dataframes from textfiles with `read.table()`, and if the data is in CSV format (that's Comma-Separated Values), we would use `read.csv()`.

We can get a column from `df$columnname`. and we can return (and change) the column names by querying `names(df)`.

We can add a column to the right with `cbind(df,newCol)`, and a row below with `rbind(df, newRow)`.