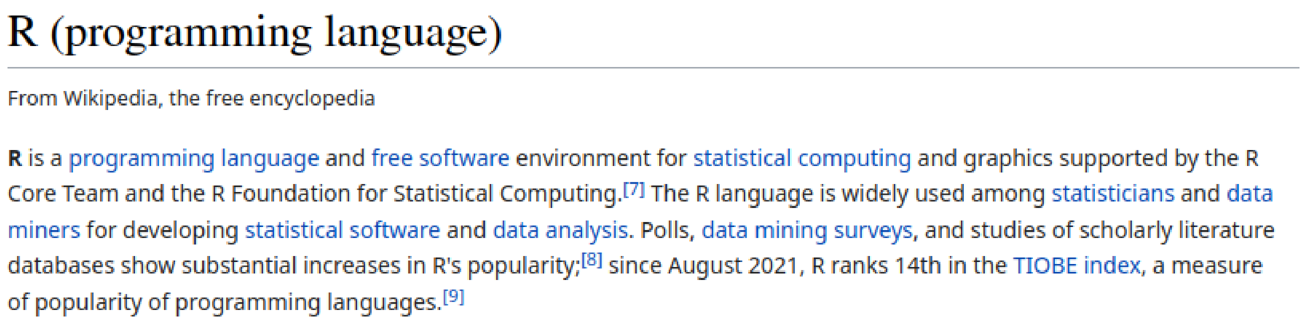
Introduction to R

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

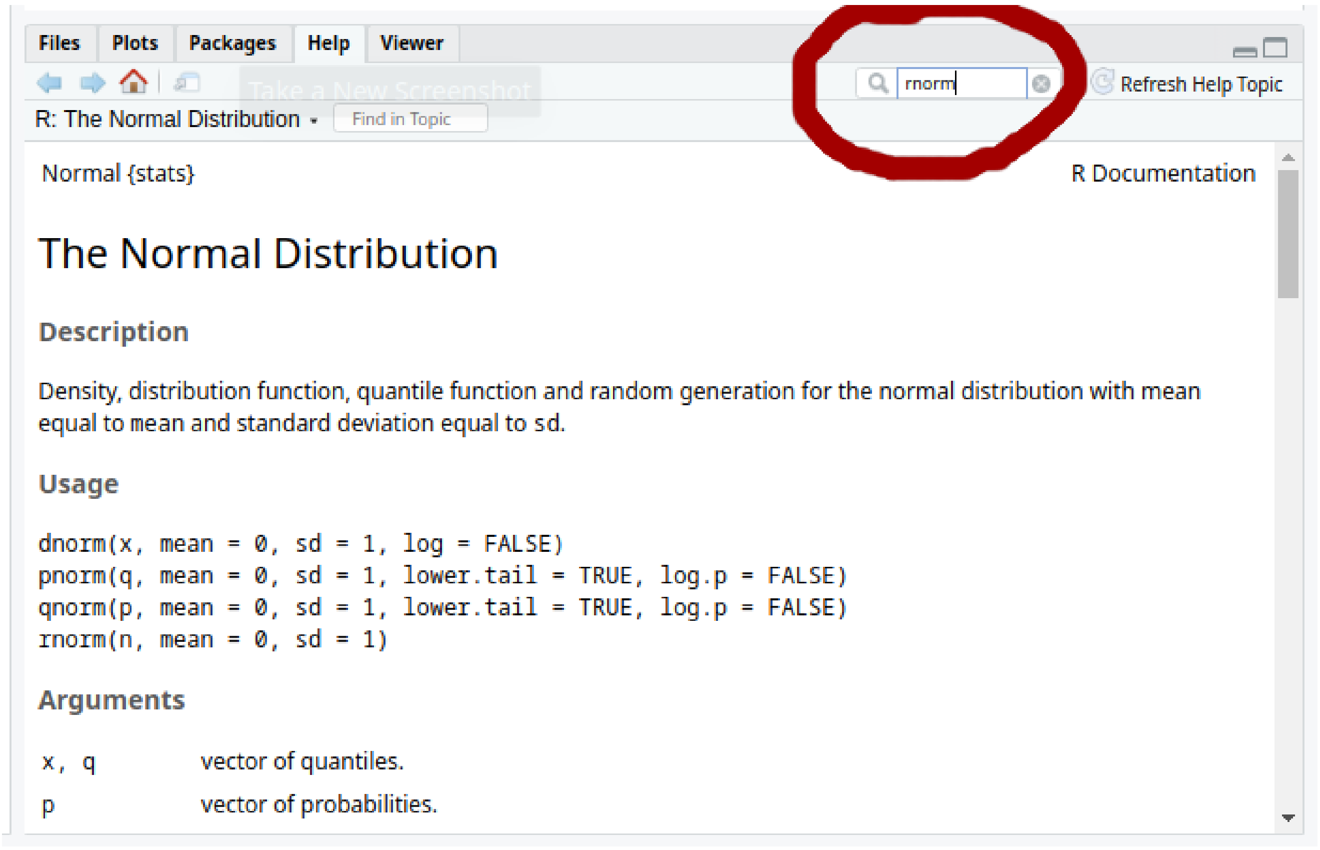
There is a command prompt

```
>
```

Here we type commands. We can ask for help:

```
help(anova)
```

If using RStudio, this opens the docpage in the window to the right.
![](assets/Pasted image 20210901113529.png)
In RStudio, we can also use the searchbar in the Help tab on the right:



Normal {stats} R Documentation

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to mean and standard deviation equal to sd.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

x, q	vector of quantiles.
p	vector of probabilities.

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...

February 26, 2003

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 them 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`.

Selecting items from arrays/matrices

Suppose that we have an array `arr` consisting of the numbers from 1 through 10.

```
> arr = 1:10
```

We can the 3rd element of the array by doing `arr[3]`, with square brackets:

```
> arr[3]
[1] 3
```

If we want to create a new array with the 1st, 5th, 8th, and 10th entry,
then we put a *list* with the indices 1, 5, 8, 10 within the square brackets:

```
> arr[c(1,5,8,10)]
[1]  1  5  8 10
```

(Question: Why did we write `c(1,5,8,10)` and not just `1,5,8,10`? Try without the `c` and see what happens.)

Now suppose that we have a long list of the numbers from 1 through 100, and we want to get from it another array with the entries that are multiples of 7. How do we do this? First, we notice that if we pass an array of *boolean* (true/false) values to the array, we get the entries that correspond to true:

```
> arr2 = c("a","b","c","d","e","f")
> arr2[c(TRUE, FALSE, FALSE, TRUE, TRUE, FALSE)]
[1] "a" "d" "e"
```

Furthermore, if we compare an *array* with a single item, we get a list of entry-by-entry comparison. For example

```
> arr3 = c(-19,27,43,0,2,-2,-8,10)
> arr3 >= 8
[1] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
```

Here we are taking each entry of `arr3` and asking whether it is >= 8 or not. Then we get an array with `TRUE` if the element is indeed >= 8, or `FALSE` if not.

Now we can combine these last ideas to extract our array of multiples of 7.

```
> arr4 = 1:100
> whichmultseven = (arr4%%7 == 0)
> arr4[whichmultseven]
[1]  7 14 21 28 35 42 49 56 63 70 77 84 91 98
```

In the first line we construct the array of numbers from 1 through 100.
In the second line, we construct an array of TRUE/FALSE values, where each value tells us if the corresponding entry of `arr4` is a multiple of 7 or not.
In the last line, we pass this array of TRUE/FALSE values to the square brackets in `arr4` to get the ones that are indeed multiples of 7.
We can condense all of this in a single line:

```
arr4[arr4%%7 == 0]
[1]  7 14 21 28 35 42 49 56 63 70 77 84 91 98
```

Now when we have matrices, we must pass two values within square brackets to get an entry back:

```
> mat1 = matrix(arr4, nrow=10)
> mat1
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   11   21   31   41   51   61   71   81    91
[2,]    2   12   22   32   42   52   62   72   82    92
[3,]    3   13   23   33   43   53   63   73   83    93
[4,]    4   14   24   34   44   54   64   74   84    94
[5,]    5   15   25   35   45   55   65   75   85    95
[6,]    6   16   26   36   46   56   66   76   86    96
[7,]    7   17   27   37   47   57   67   77   87    97
[8,]    8   18   28   38   48   58   68   78   88    98
[9,]    9   19   29   39   49   59   69   79   89    99
[10,]   10   20   30   40   50   60   70   80   90   100
> mat1[8,9]
[1] 88
```

Here we can be more creative and get *submatrices*. For example, suppose that we just want the 2nd and 3rd column and the 5th, 9th and 10th row. Then we can pass those as *arrays* in the square brackets:

```
> mat1[c(2,3),c(5,9,10)]
    [,1] [,2] [,3]
[1,]   42   82   92
[2,]   43   83   93
```

What if we want a few rows, but *all* columns? Then we leave the column index blank:

```
> mat1[c(5,9), ]
    [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    5   15   25   35   45   55   65   75   85    95
[2,]    9   19   29   39   49   59   69   79   89    99
```

Similarly, if we want just a few columns but *all* rows, we leave the row index blank:

```
> mat1[ , c(1,5,7)]
     [,1] [,2] [,3]
[1,]    1   41   61
[2,]    2   42   62
[3,]    3   43   63
[4,]    4   44   64
[5,]    5   45   65
[6,]    6   46   66
[7,]    7   47   67
[8,]    8   48   68
[9,]    9   49   69
[10,]   10   50   70
```

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”.

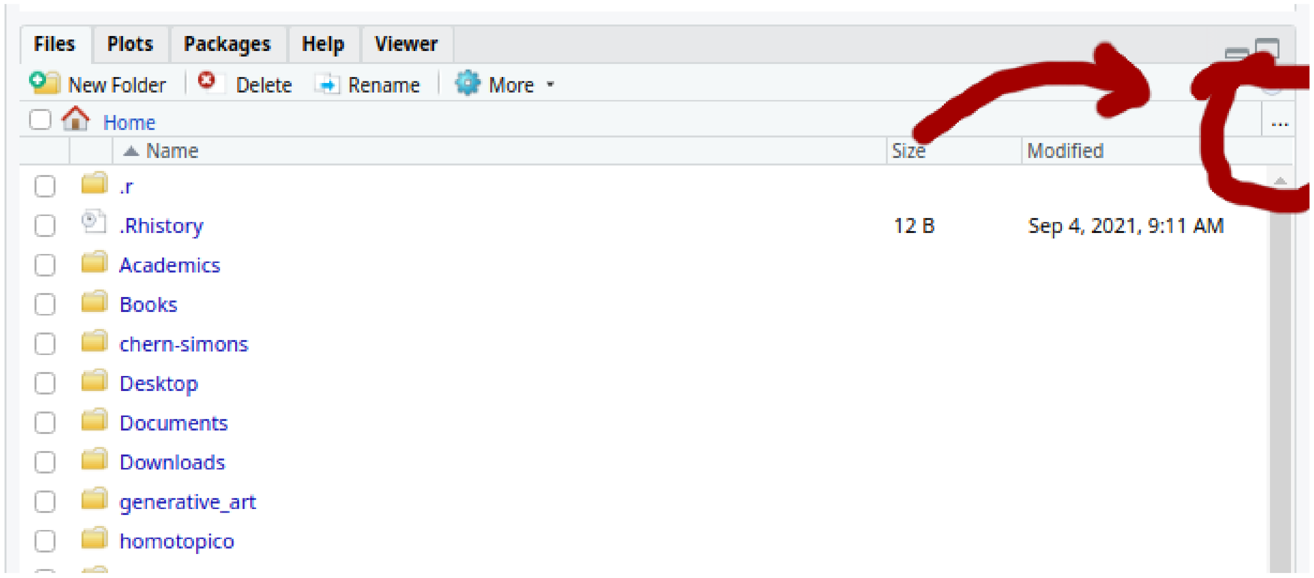
Loading data and dataframes

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.

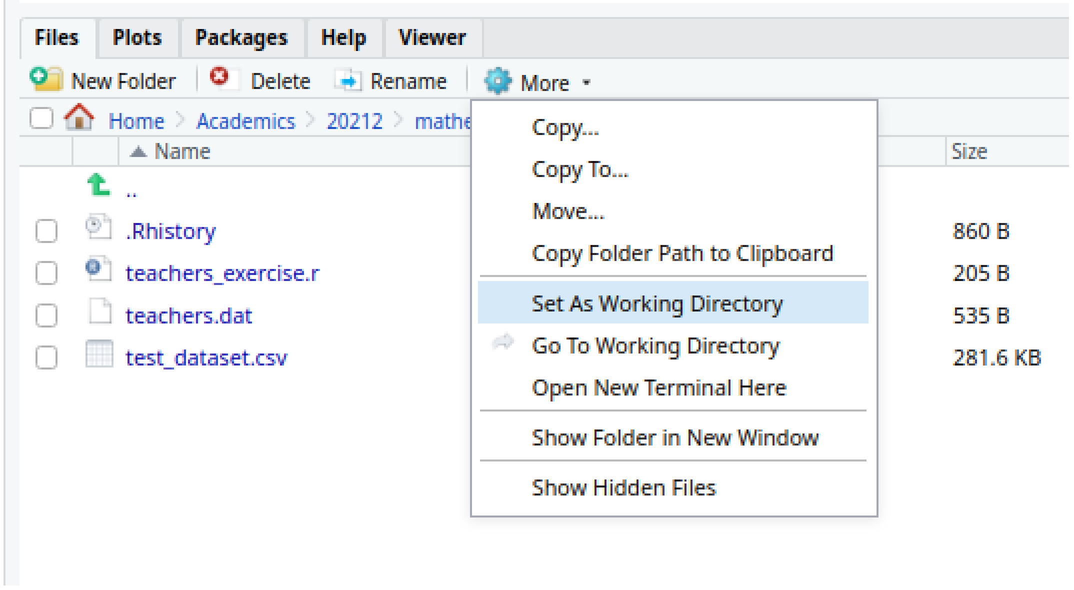
In barebones R (i.e. the command console) we can set the working directory with

```
> 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.

Now download the dataset (either from ItsLearning or from this link) and *make sure that the dataset is in the working directory*.

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()`. In this case, we are interested in the dataset `test_dataset.csv`. We load it into a variable called `dataset`.

```
> dataset = read.csv("test_dataset.csv")
```

We can check the first few rows of the dataset with the `head()` function:

```
> head(dataset)
          name pokedex_number            abilities       typing hp attack
1      Bulbasaur              1 Overgrow~Chlorophyll Grass~Poison 45     49
2        Ivysaur              2 Overgrow~Chlorophyll Grass~Poison 60     62
3       Venusaur              3 Overgrow~Chlorophyll Grass~Poison 80     82
4 Venusaur Gmax              3 Overgrow~Chlorophyll Grass~Poison 80     82
5 Venusaur Mega              3            Thick Fat Grass~Poison 80    100
6     Charmander              4     Blaze~Solar Power         Fire 39     52
```

Calling the `str()` function also gives us an overview of the dataframe:

```
> str(dataset)
'data.frame':   1118 obs. of  49 variables:
 $ name                        : chr  "Bulbasaur" "Ivysaur" "Venusaur"
"Venusaur Gmax" ...
 $ pokedex_number              : int  1 2 3 3 3 4 5 6 6 6 ...
 $ abilities                   : chr  "Overgrow~Chlorophyll"
"Overgrow~Chlorophyll" "Overgrow~Chlorophyll" "Overgrow~Chlorophyll" ...
 $ typing                      : chr  "Grass~Poison" "Grass~Poison"
"Grass~Poison" "Grass~Poison" ...
 $ hp                          : int  45 60 80 80 80 39 58 78 78 78 ...
 $ attack                      : int  49 62 82 82 100 52 64 84 84 130 ...
 $ defense                     : int  49 63 83 83 123 43 58 78 78 111 ...
 $ special_attack              : int  65 80 100 100 122 60 80 109 109 130
...
 $ special_defense             : int  65 80 100 100 120 50 65 85 85 85 ...
 $ speed                       : int  45 60 80 80 80 65 80 100 100 100 ...
 $ height                      : int  7 10 20 240 24 6 11 17 280 17 ...
 $ weight                      : int  69 130 1000 10000 1555 85 190 905
10000 1105 ...
 $ genus                       : chr  "Seed Pok\303\251mon" "Seed
Pok\303\251mon" "Seed Pok\303\251mon" "Seed Pok\303\251mon" ...
 ...
```

With this we see that this is a list of Pokémon.[1][2]
We can get a column from `df$columnname`, and we can return (and change) the column names by querying `names()`:

```
> dataset$name
  [1] Bulbasaur                    Ivysaur
  [3] Venusaur                     Venusaur Gmax
  [5] Venusaur Mega                Charmander
  [7] Charmeleon                   Charizard
  [9] Charizard Gmax               Charizard Mega X
 [11] Charizard Mega Y             Squirtle
 [13] Wartortle                    Blastoise
 [15] Blastoise Gmax               Blastoise Mega
 [17] Caterpie                     Metapod
 [19] Butterfree                   Butterfree Gmax
 [21] Weedle                       Kakuna
 [23] Beedrill                     Beedrill Mega
 [25] Pidgey                       Pidgeotto
 ...
 > names(dataset)
[1] "name"                           "pokedex_number"
[3] "abilities"                      "typing"
[5] "hp"                             "attack"
[7] "defense"                        "special_attack"
[9] "special_defense"                "speed"
[11] "height"                         "weight"
[13] "genus"                          "gen_introduced"
[15] "female_rate"                    "genderless"
[17] "baby_pokemon"                   "legendary"
[19] "mythical"                       "is_default"
[21] "forms_switchable"               "base_experience"
[23] "capture_rate"                   "egg_groups"
[25] "egg_cycles"                     "base_happiness"
[27] "can_evolve"                     "evolves_from"
[29] "primary_color"                  "shape"
[31] "number_pokemon_with_typing"     "normal_attack_effectiveness"
[33] "fire_attack_effectiveness"      "water_attack_effectiveness"
[35] "electric_attack_effectiveness" "grass_attack_effectiveness"
[37] "ice_attack_effectiveness"       "fighting_attack_effectiveness"
[39] "poison_attack_effectiveness"    "ground_attack_effectiveness"
[41] "fly_attack_effectiveness"       "psychic_attack_effectiveness"
[43] "bug_attack_effectiveness"       "rock_attack_effectiveness"
[45] "ghost_attack_effectiveness"     "dragon_attack_effectiveness"
```

```
[47] "dark_attack_effectiveness"     "steel_attack_effectiveness"
[49] "fairy_attack_effectiveness"
```

This dataset has way too many columns (and if we know the rock-paper-scissors rules between Pokémon types, we know the attack effectiveness so columns 32-49 are a lot of superfluous information), so we want to select just a few. Say that we want to keep only the first 20 columns. Thinking of this dataset as a matrix, from the above we can do

```
> dataset = dataset[ , 1:20]
```

i.e. selecting *all* rows and our specified columns.

Now suppose that we only wanted to keep the columns `name`, `hp`, and `typing`. Here we can pass a list of *column names* to the square brackets and it will work:

```
> head(dataset[ , c("name", "hp", "typing")])
           name hp        typing
1     Bulbasaur 45 Grass~Poison
2       Ivysaur 60 Grass~Poison
3      Venusaur 80 Grass~Poison
4 Venusaur Gmax 80 Grass~Poison
5 Venusaur Mega 80 Grass~Poison
6    Charmander 39          Fire
...
```

Note that the order of the columns is the same as the order of the list we passed, even though in the original dataset, `typing` goes before `hp`.

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

---

1. Did you know that Pokémon is the highest-grossing media franchise? And most of the profits come from merch.↩
2. I got this dataset from Kaggle. There's many others to play with!↩