Lecture 5: R Basics, Part 3

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Rutgers University – New Jersey Medical School

4/20/2024

Sorting

Now that we have mastered some basic R knowledge, let's try to gain some insights into the safety of different states in the context of gun murders.

Say we want to rank the states from least to most gun murders. The function `sort` sorts a vector in increasing order. We can therefore see the largest number of gun murders by typing:

```
library(dslabs)
data(murders)
sort(murders$total)
```

```
##  [1]    2    4    5    5    7    8   11   12   12   16   19   21   22   27
## [16]   36   38   53   63   65   67   84   93   93   97   97   99  111  116
## [31]  120  135  142  207  219  232  246  250  286  293  310  321  351  364
## [46]  413  457  517  669  805 1257
```

However, this does not give us information about which states have which murder totals. For example, we don't know which state had 1257 murders.

Sorting

The function `order` is closer to what we want. It takes a vector as input and returns the vector of indexes that sorts the input vector. This may sound confusing so let's look at a simple example. We can create a vector and sort it:

```
x <- c(31, 4, 15, 92, 65)
sort(x)
```

```
## [1]  4 15 31 65 92
```

Rather than sort the input vector, the function `order` returns the index that sorts input vector:

```
index <- order(x)
x[index]
```

```
## [1]  4 15 31 65 92
```

Sorting

This is the same output as that returned by `sort(x)`. If we look at this index, we see why it works:

```
x
```

```
## [1] 31  4 15 92 65
```

```
order(x)
```

```
## [1] 2 3 1 5 4
```

The second entry of `x` is the smallest, so `order(x)` starts with 2. The next smallest is the third entry, so the second entry is 3, etc.

Sorting

How does this help us order the states by murders? First, remember that the entries of vectors you access with $ follow the same order as the rows in the table. For example, these two vectors containing state names and abbreviations are matched by their order:

```
murders$state[1:6]
```

```
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"
## [6] "Colorado"
```

```
murders$abb[1:6]
```

```
## [1] "AL" "AK" "AZ" "AR" "CA" "CO"
```

Sorting

This means we can order the state names by their total murders. We first obtain the index that orders the vectors according to murder totals and then index the state names vector:

```
ind <- order(murders$total)
murders$abb[ind]
```

```
##  [1] "VT" "ND" "NH" "WY" "HI" "SD" "ME" "ID" "MT" "RI" "AK" "IA" "UT" "WV" "
## [16] "OR" "DE" "MN" "KS" "CO" "NM" "NV" "AR" "WA" "CT" "WI" "DC" "OK" "KY" "
## [31] "MS" "AL" "IN" "SC" "TN" "AZ" "NJ" "VA" "NC" "MD" "OH" "MO" "LA" "IL" "
## [46] "MI" "PA" "NY" "FL" "TX" "CA"
```

According to the above, California had the most murders.

Sorting

If we are only interested in the entry with the largest value, we can use `max` for the value:

```
max(murders$total)
```

```
## [1] 1257
```

and `which.max` for the index of the largest value:

```
i_max <- which.max(murders$total)
murders$state[i_max]
```

```
## [1] "California"
```

For the minimum, we can use `min` and `which.min` in the same way.

Sorting

Does this mean California is the most dangerous state? In an upcoming section, we argue that we should be considering rates instead of totals. Before doing that, we introduce one last order-related function: `rank`.

Although not as frequently used as `order` and `sort`, the function `rank` is also related to order and can be useful. For any given vector it returns a vector with the rank of the first entry, second entry, etc., of the input vector. Here is a simple example:

```
x <- c(31, 4, 15, 92, 65)
rank(x)
```

```
## [1] 3 1 2 5 4
```

Sorting

To summarize, let's look at the results of the three functions we have introduced:

original	sort	order	rank
31	4	2	3
4	15	3	1
15	31	1	2
92	65	5	5
65	92	4	4

Beware of recycling

Another common source of unnoticed errors in R is the use of **recycling**. We saw that vectors are added elementwise. So if the vectors don't match in length, it is natural to assume that we should get an error. But we don't. Notice what happens:

```
x <- c(1, 2, 3)
y <- c(10, 20, 30, 40, 50, 60, 70)
x+y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
## [1] 11 22 33 41 52 63 71
```

We do get a warning, but no error. For the output, R has recycled the numbers in `x`. Notice the last digit of numbers in the output.

Exercises

Now open the **R Basics Exercises** file and complete Exercises 24-31.

Vector Arithmetic

California had the most murders, but does this mean it is the most dangerous state? What if it just has many more people than any other state? We can quickly confirm that California indeed has the largest population:

```
library(dslabs)
data("murders")
murders$state[which.max(murders$population)]
```

```
## [1] "California"
```

with over 37 million inhabitants. It is therefore unfair to compare the totals if we are interested in learning how safe the state is.

Rescaling a Vector

What we really should be computing is the murders per capita. The reports we describe in the motivating section used murders per 100,000 as the unit.

To compute this quantity, the powerful vector arithmetic capabilities of R come in handy.

Rescaling a Vector

In R, arithmetic operations on vectors occur **element-wise**. For a quick example, suppose we have height in inches:

```
inches <- c(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)
```

and want to convert to centimeters. Notice what happens when we multiply `inches` by 2.54:

```
inches * 2.54
```

```
##  [1] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 18
```

Rescaling a Vector

In the previous slide, we multiplied each element by 2.54. Similarly, if for each entry we want to compute how many inches taller or shorter than 69 inches, the average height for males, we can subtract it from every entry like this:

```
inches - 69
```

```
##  [1]  0 -7 -3  1  1  4 -2  4 -2  1
```

Two vectors

If we have two vectors of the same length, and we sum them in R, they will be added entry by entry as follows:

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} + \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} a+e \\ b+f \\ c+g \\ d+h \end{pmatrix}$$

Two vectors

The same holds for other operations, such as `-`, `*` and `/`.

This implies that to compute the murder rates we can simply type:

```
murder_rate <- murders$total / murders$population * 100000
```

Two vectors

Once we do this, we notice that California is no longer near the top of the list. In fact, we can use what we have learned to order the states by murder rate:

```
murders$abb[order(murder_rate)]
```

```
##  [1] "VT" "NH" "HI" "ND" "IA" "ID" "UT" "ME" "WY" "OR" "SD" "MN" "MT" "CO" "
## [16] "WV" "RI" "WI" "NE" "MA" "IN" "KS" "NY" "KY" "AK" "OH" "CT" "NJ" "AL" "
## [31] "OK" "NC" "NV" "VA" "AR" "TX" "NM" "CA" "FL" "TN" "PA" "AZ" "GA" "MS" "
## [46] "DE" "SC" "MD" "MO" "LA" "DC"
```

Exercises

Now open the **R Basics Exercises** file and complete Exercises 32-34.

Indexing

R provides a powerful and convenient way of indexing vectors. We can, for example, subset a vector based on properties of another vector. Imagine you are moving from Italy where, according to an ABC news report, the murder rate is only 0.71 per 100,000.

You would prefer to move to a state with a similar murder rate. Another powerful feature of R is that we can use logicals to index vectors. If we compare a vector to a single number, it actually performs the test for each entry. The following is an example related to the question above:

```
ind <- murder_rate < 0.71
```

Indexing

If we instead want to know if a value is less or equal, we can use:

```
ind <- murder_rate <= 0.71
```

Note that we get back a logical vector with `TRUE` for each entry smaller than or equal to 0.71. To see which states these are, we can leverage the fact that vectors can be indexed with logicals.

```
murders$state[ind]
```

```
## [1] "Hawaii"        "Iowa"          "New Hampshire" "North Dakota"
## [5] "Vermont"
```

Indexing

In order to count how many are TRUE, the function `sum` returns the sum of the entries of a vector and logical vectors get *coerced* to numeric with `TRUE` coded as 1 and `FALSE` as 0. Thus we can count the states using:

```
sum(ind)
```

```
## [1] 5
```

Logical operators

Suppose we like the mountains and we want to move to a safe state in the western region of the country. We want the murder rate to be at most 1. In this case, we want two different things to be true. Here we can use the logical operator *and*, which in R is represented with `&`. This operation results in `TRUE` only when both logicals are `TRUE`. To see this, consider this example:

```
TRUE & TRUE
```

```
## [1] TRUE
```

```
TRUE & FALSE
```

```
## [1] FALSE
```

```
FALSE & FALSE
```

```
## [1] FALSE
```

Logical operators

For our example, we can form two logicals:

```
west <- murders$region == "West"
safe <- murder_rate <= 1
```

and we can use the & to get a vector of logicals that tells us which states satisfy both conditions:

```
ind <- safe & west
murders$state[ind]
```

```
## [1] "Hawaii"  "Idaho"   "Oregon"  "Utah"    "Wyoming"
```

Logical operators: `which`

Suppose we want to look up California's murder rate. For this type of operation, it is convenient to convert vectors of logicals into indexes instead of keeping long vectors of logicals. The function `which` tells us which entries of a logical vector are TRUE. So we can type:

```
ind <- which(murders$state == "California")
murder_rate[ind]
```

```
## [1] 3.374138
```

Logical operators: match

If instead of just one state we want to find out the murder rates for several states, say New York, Florida, and Texas, we can use the function match. This function tells us which indexes of a second vector match each of the entries of a first vector:

```
ind <- match(c("New York", "Florida", "Texas"), murders$state)
ind
```

```
## [1] 33 10 44
```

Now we can look at the murder rates:

```
murder_rate[ind]
```

```
## [1] 2.667960 3.398069 3.201360
```

Logical operators: %in%

If rather than an index we want a logical that tells us whether or not each element of a first vector is in a second, we can use the function %in%. Let's imagine you are not sure if Boston, Dakota, and Washington are states. You can find out like this:

```
c("Boston", "Dakota", "Washington") %in% murders$state
```

```
## [1] FALSE FALSE  TRUE
```

Note that we will be using %in% often throughout this tutorial.

Logical operators: %in%

Advanced: There is a connection between `match` and `%in%` through `which`. To see this, notice that the following two lines produce the same index (although in different order):

```
match(c("New York", "Florida", "Texas"), murders$state)
```

```
## [1] 33 10 44
```

```
which(murders$state%in%c("New York", "Florida", "Texas"))
```

```
## [1] 10 33 44
```

Exercises

Now open the **R Basics Exercises** file and complete Exercises 35-42.

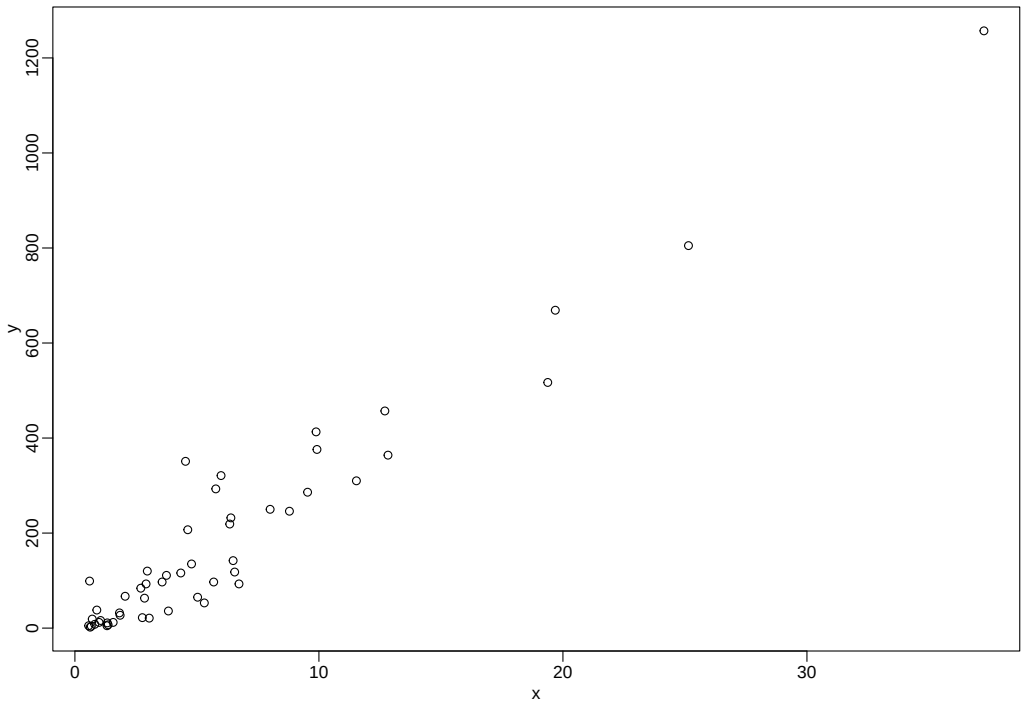
Basic Plots

Later we will present add-on package named `ggplot2` that provides a powerful approach to producing plots in R. We then have an entire part on Data Visualization in which we provide many examples. Here we briefly describe some of the functions that are available in a basic R installation.

Basic Plots: plot

The plot function can be used to make scatterplots. Here is a plot of total murders versus population.

```
x <- murders$population / 10^6
y <- murders$total
plot(x, y)
```



Basic Plots: plot

For a quick plot that avoids accessing variables twice, we can use the with function:

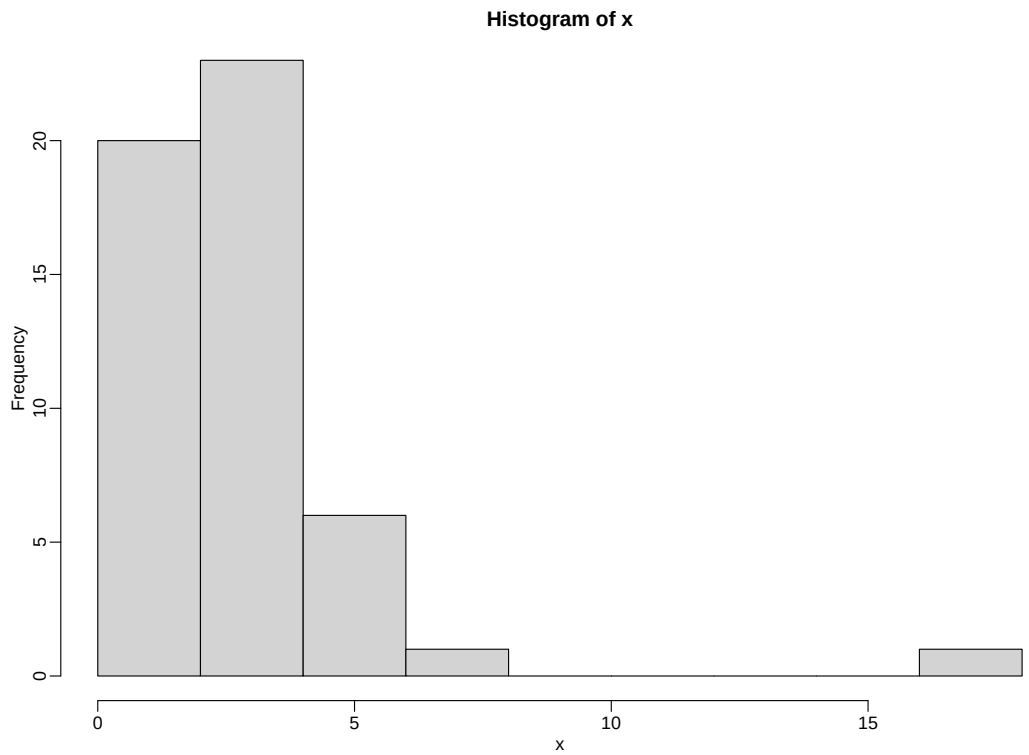
```
with(murders, plot(population, total))
```

The function with lets us use the murders column names in the plot function. It also works with any data frames and any function.

Basic Plots: hist

We will describe histograms as they relate to distributions in the Data Visualization section later. We can make a histogram of our murder rates by simply typing:

```
x <- with(murders, total / population * 100000)
hist(x)
```



Basic Plots: hist

We can see that there is a wide range of values with most of them between 2 and 3 and one very extreme case with a murder rate of more than 15:

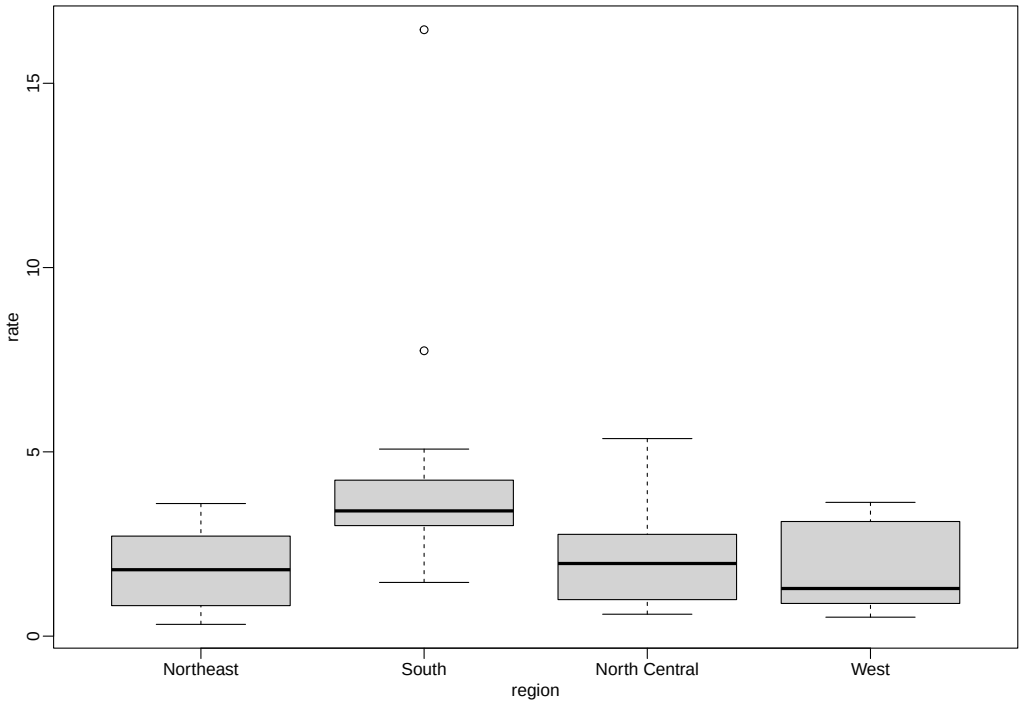
```
murders$state[which.max(x)]
```

```
## [1] "District of Columbia"
```

Basic Plots: boxplot

Boxplots will also be described in the Data Visualization part of the tutorial. They provide a more terse summary than histograms, but they are easier to stack with other boxplots. For example, here we can use them to compare the different regions:

```
murders$rate <- with(murders, total / population * 100000)
boxplot(rate~region, data = murders)
```

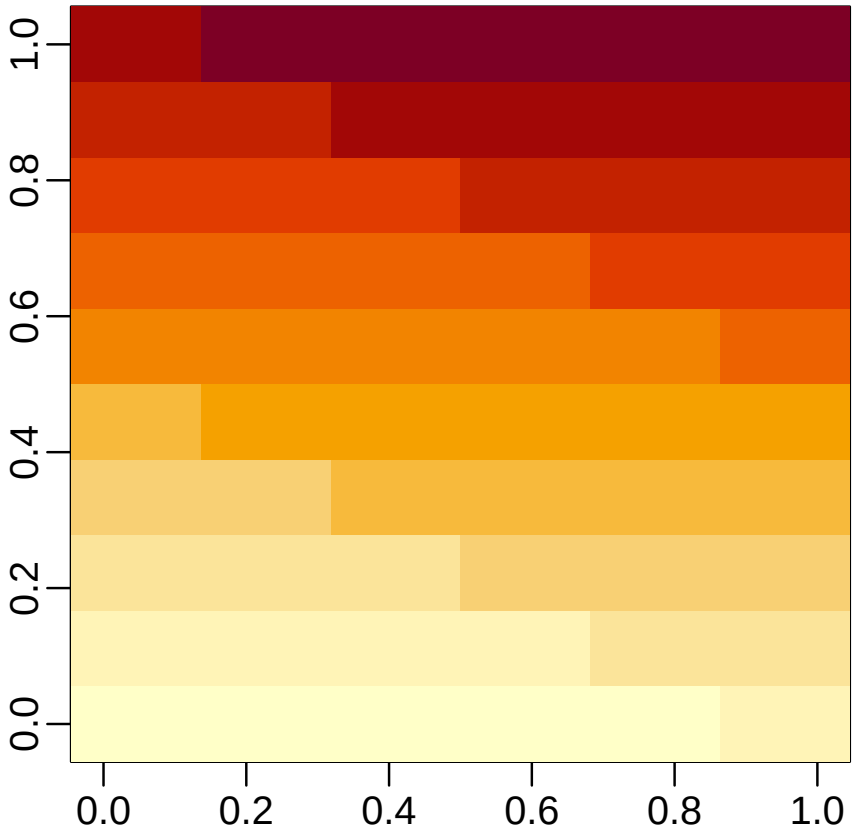


The South has higher murder rates than the other three regions.

Basic Plots: image

The image function displays the values in a matrix using color. Here is a quick example:

```
x <- matrix(1:120, 12, 10)
image(x)
```



Exercises

Now open the **R Basics Exercises** file and complete Exercises 43-45.

Session Info

```
sessionInfo()
```

```
## R version 4.3.2 (2023-10-31)
## Platform: aarch64-apple-darwin20 (64-bit)
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRlapack.dylib;  LAPACK vers
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] dslabs_0.7.6      countrycode_1.5.0 ggflags_0.0.4     lubridate_1.9.3
##  [5] forcats_1.0.0     stringr_1.5.1     dplyr_1.1.4       purrr_1.0.2
##  [9] readr_2.1.5       tidyr_1.3.1       tibble_3.2.1      ggplot2_3.5.0
## [13] tidyverse_2.0.0
##
## loaded via a namespace (and not attached):
##  [1] utf8_1.2.4          generics_0.1.3      xml2_1.3.6          stringi_1.8.3
##  [5] hms_1.1.3           digest_0.6.35       magrittr_2.0.3      evaluate_0.23
##  [9] grid_4.3.2          timechange_0.3.0    RColorBrewer_1.1-3  fastmap_1.1.1
## [13] fansi_1.0.6         viridisLite_0.4.2   scales_1.3.0        cli_3.6.2
## [17] rlang_1.1.3         munsell_0.5.1       withr_3.0.0         yaml_2.3.8
## [21] tools_4.3.2         tzdb_0.4.0          colorspace_2.1-0    kableExtra_1.4.0
## [25] vctrs_0.6.5         R6_2.5.1            lifecycle_1.0.4     pkgconfig_2.0.3
## [29] pillar_1.9.0        gtable_0.3.4        glue_1.7.0          systemfonts_1.0.6
## [33] xfun_0.43           tidyselect_1.2.1    rstudioapi_0.16.0   knitr_1.45
```