# R Basics

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

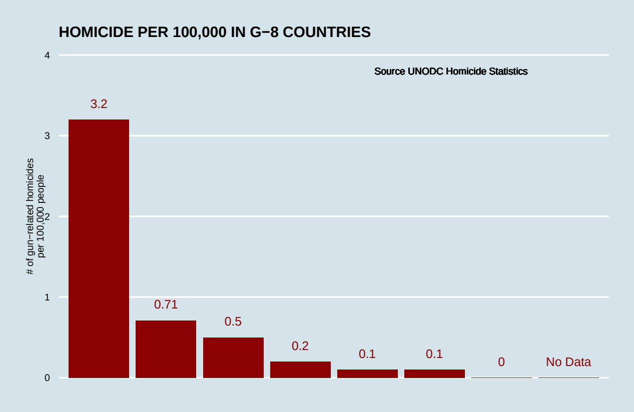
2025-02-01

WEJ LAB

Center for Data Science

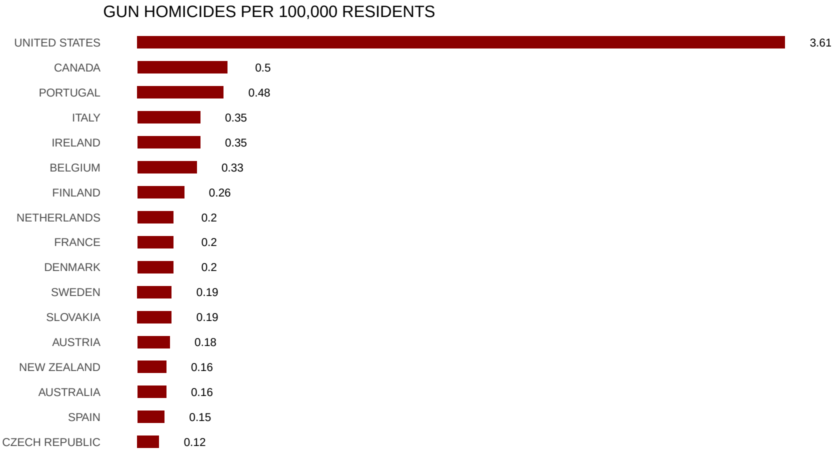
# Case study: US Gun Murders

Imagine you are trying to use a data-driven approach to deciding your opinion on gun regulations. In the process you notice the following trends:



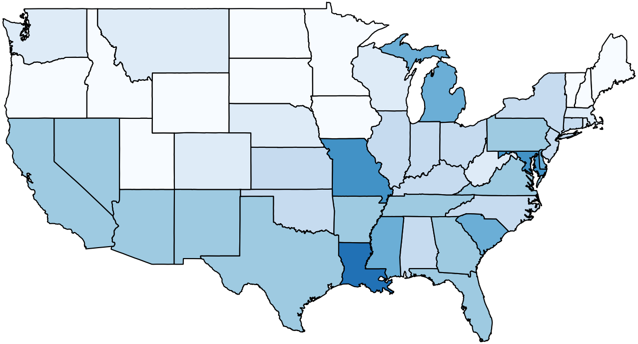
# Case study: US Gun Murders

Or even worse, this version from everytown.org:



# Case study: US Gun Murders

But then you remember that the US is a large and diverse country with 50 very different states:



# Case study: US Gun Murders

California, for example, has a larger population than Canada, and 20 US states have populations larger than that of Norway.

Furthermore, the murder rates in Lithuania, Ukraine, and Russia (not included) are higher than 4 per 100,000. So how does this factor into your decision making process?

We will gain some insights by examining data related to gun homicides in the US during 2010 using R.

# The very basics of R

But before we get started with our example, lets cover some very basic building blocks for R programming.

# Objects

Suppose we wanted to solve quadratic equations of the form $ax^2 + bx + c = 0$. The quadratic formula gives us the solutions:

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

which depends on the values of $a$, $b$, and $c$. A programming language can be use to define variables and write expressions with these variables, similar to how we do so in math, but obtain a numeric solution.

# Objects

We will write out general code for the quadratic equation below, but if we are asked to solve $x^2 + x - 1 = 0$, then we define:

```
a <- 1
b <- 1
c <- -1
```

which stores the values for later use. We use <- to assign values to the variables. We can also assign values using = instead of <-, but we recommend against using = to avoid confusion.

# Objects

To see the value stored in a variable, we simply ask R to evaluate `a` and it shows the stored value:

```r
a
```

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

A more explicit way to ask R to show us the value stored in `a` is using `print` like this:

```r
print(a)
```

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

# Objects

We use the term **object** to describe stuff that is stored in R. Variables are examples, but objects can also be more complicated entities such as functions, which are described later.

# The Workspace

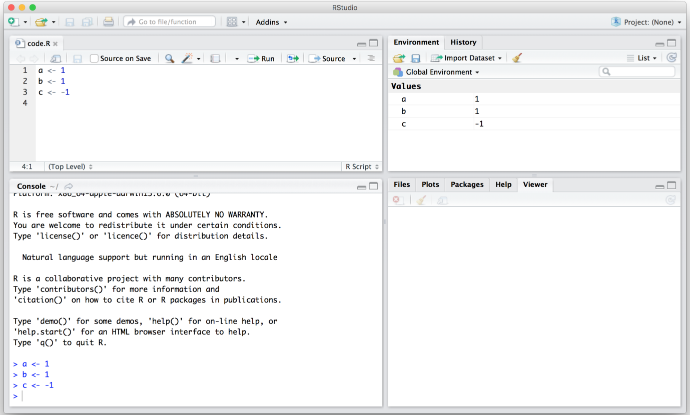
As we define objects in the console, we are changing the **workspace**. You can see all the variables saved in your workspace by typing:

```r
ls()
```

```
## [1] "a"        "b"        "c"        "dat"      "img_path" "murders"
```

# The workspace

In RStudio, the **Environment** tab shows the values:



# The Workspace

We should see `a`, `b`, and `c`. If you try to recover the value of a variable that is not in your workspace, you receive an error. For example, if you type `x` you will receive the following message: `Error: object 'x' not found`.

# The workspace

Now since these values are saved in variables, to obtain a solution to our equation, we use the quadratic formula:

```
(-b + sqrt(b^2 - 4*a*c) ) / ( 2*a )
```

```
## [1] 0.618034
```

```
(-b - sqrt(b^2 - 4*a*c) ) / ( 2*a )
```

```
## [1] -1.618034
```

# Functions

Once you define variables, the data analysis process can usually be described as a series of **functions** applied to the data. R includes several predefined functions and most of the analysis pipelines we construct make extensive use of these.

# Functions

We already used the `install.packages`, `library`, and `ls` functions. We also used the function `sqrt` to solve the quadratic equation above.

There are many more prebuilt functions and even more can be added through packages. These functions do not appear in the workspace because you did not define them, but they are available for immediate use.

# Functions

In general, we need to use parentheses to evaluate a function. If you type `ls`, the function is not evaluated and instead R shows you the code that defines the function. If you type `ls()` the function is evaluated and, as seen above, we see objects in the workspace.

# Functions

Unlike `ls`, most functions require one or more **arguments**. Below is an example of how we assign an object to the argument of the function `log`. Remember that we earlier defined `a` to be 1:

```
log(8)
```

```
## [1] 2.079442
```

```
log(a)
```

```
## [1] 0
```

# Functions

You can find out what the function expects and what it does by reviewing the very useful manuals included in R. You can get help by using the `help` function like this:

```
help("log")
```

For most functions, we can also use this shorthand:

```
?log
```

# Functions

The help page will show you what arguments the function is expecting. For example, `log` needs `x` and `base` to run.

However, some arguments are required and others are optional. You can determine which arguments are optional by noting in the help document that a default value is assigned with =. For example, the base of the function `log` defaults to `base = exp(1)` making `log` the natural log by default.

# Functions

If you want a quick look at the arguments without opening the help system, you can type:

```r
args(log)
```

```
## function (x, base = exp(1))
## NULL
```

# Functions

You can change the default values by simply assigning another object:

```r
log(8, base = 2)
```

```
## [1] 3
```

Note that we have not been specifying the argument `x` as such:

```r
log(x = 8, base = 2)
```

```
## [1] 3
```

# Functions

The above code works, but we can save ourselves some typing the following:

```r
log(8,2)
```

```
## [1] 3
```

R assumes you are entering arguments in the order shown in the help file or by `args`. So by not using the names, it assumes the arguments are `x` followed by `base`.

# Functions

If using the arguments' names, then we can include them in whatever order we want:

```r
log(base = 2, x = 8)
```

```
## [1] 3
```

To specify arguments, we must use =, and cannot use <-.

# Functions

There are some exceptions to the parentheses rule functions. Among these, the most commonly used are the arithmetic and relational operators. For example:

```r
2 ^ 3
```

```
## [1] 8
```

# Functions

You can see the arithmetic operators by typing:

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

or

```
?"+"
```

and the relational operators by typing:

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

or

```
?">"
```

# Other Prebuilt Objects

There are several datasets that are included for users to practice and test out functions. You can see all the available datasets by typing:

```
data()
```

This shows you the object name for these datasets. These datasets are objects that can be used by simply typing the name. For example, if you type:

```
co2
```

R will show you Mauna Loa atmospheric CO2 concentration data that is prebuilt into R.

# Other Prebuilt Objects

Other prebuilt objects are mathematical quantities, such as the constant $\pi$ and $\infty$:

```
pi
```

```
## [1] 3.141593
```

```
Inf+1
```

```
## [1] Inf
```

# Variable Names

We have used the letters `a`, `b`, and `c` as variable names, but variable names can be almost anything. Some basic rules in R for variable names:

1. They have to start with a letter
2. They can't contain spaces
3. They should not be variables predefined in R.

For example, for the third point, don't name one of your variables:

```
install.packages <- 2
```

which will overwrite the `install.packages` function in your workspace and you can no longer use it.

# Variable Names

A nice convention to follow:

1. Use meaningful words that describe what is stored
2. Use only lower case
3. Use underscores as a substitute for spaces

For the quadratic equations, we could use something like this:

```
solution_1 <- (-b + sqrt(b^2 - 4*a*c)) / (2*a)
solution_2 <- (-b - sqrt(b^2 - 4*a*c)) / (2*a)
```

For more advice, we highly recommend studying Hadley Wickham's style guide[1].

[1] http://adv-r.had.co.nz/Style.html

# Saving your workspace

Values remain in the workspace until you end your session or erase them with the function `rm`, but whole workspaces can also can be saved.

In fact, when you quit R, the program asks you if you want to save your workspace. If you do save it, the next time you start R, the program will restore the workspace.

# Saving your workspace

In most cases, you should avoid automatically saving the workspace. As you start working on different projects, it will become harder to keep track of what is saved. Instead, we recommend you assign the workspace a specific name.

You can save your workspace using `save` or `save.image`, and reload it using `load`. We recommend the suffix `rda` or `RData`.

In RStudio, you can also do this by navigating to the **Session** tab and choosing **Save Workspace as**. You can later load it using the **Load Workspace** options in the same tab.

# Motivating Scripts

To solve another equation such as $3x^2 + 2x - 1$, we can copy and paste the code above and then redefine the variables and recompute the solution:

```r
a <- 3
b <- 2
c <- -1
(-b + sqrt(b^2 - 4*a*c)) / (2*a)
(-b - sqrt(b^2 - 4*a*c)) / (2*a)
```

# Motivating Scripts

By creating and saving a script with the code above, we would not need to retype everything each time and, instead, simply change the variable names. Try writing the script above into an editor and notice how easy it is to change the variables and receive an answer.

# Commenting your code

If a line of R code starts with the symbol #, it is not evaluated. We can use this to write reminders of why we wrote particular code. For example, in the script above we could add:

```
## Code to compute solution to quadratic equation of
## the form ax^2 + bx + c
## First define the variables
a <- 3
b <- 2
c <- -1

## Now compute the solution
(-b + sqrt(b^2 - 4*a*c)) / (2*a)
(-b - sqrt(b^2 - 4*a*c)) / (2*a)
```

# Data Types

Variables in R can be of different types. For example: numbers, character strings, tables simple lists. The function `class` helps us determine what type of object we have:

```
a <- 2
class(a)
```

```
## [1] "numeric"
```

To work efficiently in R, it is important to learn the different types of variables and what we can do with these.

# Data Frames

Up to now, the variables we have defined are just one number, which is not very useful for storing data. The most common way of storing a dataset in R is in a **data frame**.

Conceptually, we can think of a data frame as a table with rows representing observations and the different variables reported for each observation defining the columns.

Data frames are particularly useful for datasets because we can combine different data types into one object.

# Data Frames

A large proportion of data analysis use data stored in a data frame.

For example, we stored the data for our motivating example in a data frame. You can access this dataset by loading the **dslabs** library and loading the `murders` dataset using the `data` function:

```
library(dslabs)
data(murders)
```

# Data Frames

To see that this is in fact a data frame, we type:

```
class(murders)
```

```
## [1] "data.frame"
```

# Examining an Object

The function `str` is useful for finding out more about the structure of an object:

```
str(murders)
```

```
## 'data.frame':    51 obs. of  5 variables:
## $ state : chr "Alabama" "Alaska" "Arizona" "Arkansas" ...
## $ abb : chr "AL" "AK" "AZ" "AR" ...
## $ region : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2
## $ population: num 4779736 710231 6392017 2915918 37253956 ...
## $ total : num 135 19 232 93 1257 ...
```

This tells us much more about the object. We see that the table has 51 rows (50 states plus DC) and five variables.

# Data Frames

We can show the first six lines using the function `head`:

```
head(murders)
```

```
##         state abb region population total
## 1     Alabama  AL  South    4779736   135
## 2      Alaska  AK   West     710231    19
## 3     Arizona  AZ   West    6392017   232
## 4    Arkansas  AR  South    2915918    93
## 5  California  CA   West   37253956  1257
## 6    Colorado  CO   West    5029196    65
```

# Data Frames

In this dataset, each state is considered an observation and five variables are reported for each state.

Before we go any further in answering our original question about different states, let's learn more about the components of this object.

# The Accessor: $

For our analysis, we will need to access the different variables represented by columns included in this data frame. To do this, we use the accessor operator $ in the following way:

```
murders$population
```

```
##  [1]  4779736   710231  6392017  2915918 37253956  5029196  3574097   897934
##  [9]   601723 19687653  9920000  1360301  1567582 12830632  6483802  3046355
## [17]  2853118  4339367  4533372  1328361  5773552  6547629  9883640  5303925
## [25]  2967297  5988927   989415  1826341  2700551  1316470  8791894  2059179
## [33] 19378102  9535483   672591 11536504  3751351  3831074 12702379  1052567
## [41]  4625364   814180  6346105 25145561  2763885   625741  8001024  6724540
## [49]  1852994  5686986   563626
```

# The Accessor: $

But how did we know to use `population`? Previously, by applying the function `str` to the object `murders`, we revealed the names for each of the five variables stored in this table. We can quickly access the variable names using:

```r
names(murders)
```

```
## [1] "state"      "abb"        "region"     "population" "total"
```

# The Accessor: $

**Important:** Note the order of the entries in `murders$population` preserves the order of the rows in our data table. This will later permit us to manipulate one variable based on the results of another. For example, we will be able to order the state names by the number of murders.

**Pro Tip**: R comes with a very nice auto-complete functionality that saves us the trouble of typing out all the names. Try typing `murders$p` then hitting the **tab** key on your keyboard. This functionality and many other useful auto-complete features are available when working in RStudio.

# Vectors: Numerics, Characters, and Logical

The object `murders$population` is not one number but several. We call these types of objects **vectors**. A single number is technically a vector of length 1, but in general we use the term vectors to refer to objects with several entries. The function `length` tells you how many entries are in the vector:

```
pop <- murders$population
length(pop)
```

```
## [1] 51
```

# Vectors: Numerics, Characters, and Logical

This particular vector is **numeric** since population sizes are numbers:

```
class(pop)
```

```
## [1] "numeric"
```

In a numeric vector, every entry must be a number.

# Vectors: Numerics, Characters, and Logical

To store character strings, vectors can also be of class **character**. For example, the state names are characters:

```
class(murders$state)
```

```
## [1] "character"
```

As with numeric vectors, all entries in a character vector need to be a character.

# Vectors: Numerics, Characters, and Logical

Another important type of vectors are **logical vectors**. These must be either TRUE or FALSE.

```r
z <- 3 == 2
z
```

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

```r
class(z)
```

```
## [1] "logical"
```

Here the == is a relational operator asking if 3 is equal to 2. In R, if you just use one =, you actually assign a variable, but if you use two == you test for equality.

# Vectors: Numerics, Characters, and Logical

You can see the other **relational operators** by typing:

```
?Comparison
```

In the future, you will see how useful relational operators can be. We discuss more important features of vectors later.

# Vectors: Numerics, Characters, and Logical

**Advanced**: Mathematically, the values in `pop` are integers and there is an integer class in R. However, by default, numbers are assigned class numeric even when they are round integers. For example, `class(1)` returns numeric. You can turn them into class integer with the `as.integer()` function or by adding an `L` like this: `1L`. Note the class by typing: `class(1L)`

# Factors

In the `murders` dataset, we might expect the region to also be a character vector. However, it is not:

```r
class(murders$region)
```

```
## [1] "factor"
```

It is a **factor**. Factors are useful for storing categorical data.

# Factors

We can see that there 4 regions by using the `levels` function:

```
levels(murders$region)
```

```
## [1] "Northeast"     "South"         "North Central" "West"
```

# Factors

In the background, R stores these **levels** as integers and keeps a map to keep track of the labels. This is more memory efficient than storing all the characters.

Note that the levels have an order that is different from the order of appearance in the factor object. The default in R is for the levels to follow alphabetical order. However, often we want the levels to follow a different order.

You can specify an order through the `levels` argument when creating the factor with the `factor` function. For example, in the murders dataset regions are ordered from east to west. The function `reorder` lets us change the order of the levels of a factor variable based on a summary computed on a numeric vector.

# Factors

Suppose we want the levels of the region by the total number of murders rather than alphabetical order. If there are values associated with each level, we can use the `reorder` and specify a data summary to determine the order. The following code takes the sum of the total murders in each region, and reorders the factor following these sums.

```
region <- murders$region
value <- murders$total
region <- reorder(region, value, FUN = sum)
levels(region)
```

```
## [1] "Northeast"     "North Central" "West"          "South"
```

# Factors

The new order is in agreement with the fact that the Northeast has the least murders and the South has the most.

**Warning**: Factors can be a source of confusion since sometimes they behave like characters and sometimes they do not. As a result, confusing factors and characters are a common source of bugs.

# Lists

Data frames are a special case of **lists**. Lists are useful because you can store any combination of different types. You can create a list using the `list` function like this:

```
record <- list(name = "John Doe",
               student_id = 1234,
               grades = c(95, 82, 91, 97, 93),
               final_grade = "A")
```

The function `c` (for concatenate) is described later.

# Lists

The list on the prior slide includes a character, a number, a vector with five numbers, and another character.

```r
class(record)
```

```
## [1] "list"
```

```r
record
```

```
## $name
## [1] "John Doe"
##
## $student_id
## [1] 1234
##
## $grades
## [1] 95 82 91 97 93
##
## $final_grade
## [1] "A"
```

# Lists

As with data frames, you can extract the components of a list with the accessor $.

```
record$student_id
```

```
## [1] 1234
```

# Lists

We can also use double square brackets (`[[`) like this:

```
record[["student_id"]]
```

```
## [1] 1234
```

You should get used to the fact that in R, there are often several ways to do the same thing, such as accessing entries.

# Lists

You might also encounter lists without variable names.

```r
record2 <- list("John Doe", 1234)
record2
```

```
## [[1]]
## [1] "John Doe"
##
## [[2]]
## [1] 1234
```

# Lists

If a list does not have names, you cannot extract the elements with $, but you can still use the brackets method and instead of providing the variable name, you provide the list index, like this:

```
record2[[1]]
```

```
## [1] "John Doe"
```

We won't be using lists until later, but you might encounter one in your own exploration of R. For this reason, we showed you some basics here.

# Matrices

**Matrices** are another type of object that are common in R. Matrices are similar to data frames in that they are two-dimensional: they have rows and columns. However, like numeric, character and logical vectors, entries in matrices have to be all the same type. For this reason data frames are much more useful for storing data, since we can have characters, factors, and numbers in them.

# Matrices

Yet matrices have a major advantage over data frames: we can perform matrix algebra operations, a powerful type of mathematical technique. We will cover matematical computions on matrices in more detail later!

# Matrices

We can define a matrix using the `matrix` function. We need to specify the number of rows and columns.

```
mat <- matrix(1:12, 4, 3)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

# Matrices

You can access specific entries in a matrix using square brackets ( [). If you want the second row, third column, you use:

```r
mat[2, 3]
```

```
## [1] 10
```

# Matrices

If you want the entire second row, you leave the column spot empty:

```
mat[2, ]
```

```
## [1]  2  6 10
```

Notice that this returns a vector, not a matrix.

# Matrices

Similarly, if you want the entire third column, you leave the row spot empty:

```
mat[, 3]
```

```
## [1]  9 10 11 12
```

This is also a vector, not a matrix.

# Matrices

You can access more than one column or more than one row if you like. This will give you a new matrix.

```
mat[, 2:3]
```

```
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
## [3,]    7   11
## [4,]    8   12
```

# Matrices

You can subset both rows and columns:

```
mat[1:2, 2:3]
```

```
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
```

# Matrices

We can convert matrices into data frames using the function `as.data.frame`:

```r
as.data.frame(mat)
```

```
##   V1 V2 V3
## 1  1  5  9
## 2  2  6 10
## 3  3  7 11
## 4  4  8 12
```

# Matrices

You can also use single square brackets ( [) to access rows and columns of a data frame:

```r
data("murders")
murders[25, 1]
```

```
## [1] "Mississippi"
```

```r
murders[2:3, ]
```

```
##     state abb region population total
## 2  Alaska  AK   West     710231    19
## 3 Arizona  AZ   West    6392017   232
```

# Exercises

Now open the **R Basics Exercises** file and complete Exercises 6-11.

# Vectors

In R, the most basic objects available to store data are **vectors**. As we have seen, complex datasets can usually be broken down into components that are vectors. For example, in a data frame, each column is a vector. Here we learn more about this important class.

# Creating Vectors

We can create vectors using the function c, which stands for **concatenate**. We use c to concatenate entries in the following way:

```
codes <- c(380, 124, 818)
codes
```

```
## [1] 380 124 818
```

# Creating Vectors

We can also create character vectors. We use the quotes to denote that the entries are characters rather than variable names.

```
country <- c("italy", "canada", "egypt")
```

# Creating Vectors

In R you can also use single quotes:

```
country <- c('italy', 'canada', 'egypt')
```

But be careful not to confuse the single quote ' with the **back quote** `.

# Creating Vectors

By now you should know that if you type:

```
country <- c(italy, canada, egypt)
```

you receive an error because the variables `'italy'`, `'canada'`, and `'egypt'` are not defined. If we do not use the quotes, R looks for variables with those names and returns an error.

# Names

Sometimes it is useful to name the entries of a vector. For example, when defining a vector of country codes, we can use the names to connect the two:

```
codes <- c(italy = 380, canada = 124, egypt = 818)
codes
```

```
##   italy canada  egypt
##     380    124    818
```

# Names

The object codes continues to be a numeric vector:

```
class(codes)
```

```
## [1] "numeric"
```

but with names:

```
names(codes)
```

```
## [1] "italy"  "canada" "egypt"
```

# Names

If the use of strings without quotes looks confusing, know that you can use the quotes as well:

```
codes <- c("italy" = 380, "canada" = 124, "egypt" = 818)
codes
```

```
##  italy canada  egypt
##    380    124    818
```

There is no difference between this function call and the previous one. This is one of the many ways in which R is quirky compared to other languages.

# Names

We can also assign names using the `names` functions:

```
codes <- c(380, 124, 818)
country <- c("italy","canada","egypt")
names(codes) <- country
codes
```

```
##   italy canada  egypt
##     380    124    818
```

# Sequences

Another useful function for creating vectors generates sequences:

```r
seq(1, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

# Sequences

The first argument defines the start, and the second defines the end which is included. The default is to go up in increments of 1, but a third argument lets us tell it how much to jump by:

```
seq(1, 10, 2)
```

```
## [1] 1 3 5 7 9
```

# Sequences

If we want consecutive integers, we can use the following shorthand:

```r
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

WEJ LAB

Center for Data Science

# Sequences

When we use these functions, R produces integers, not numerics, because they are typically used to index something:

```
class(1:10)
```

```
## [1] "integer"
```

However, if we create a sequence including non-integers, the class changes:

```
class(seq(1, 10, 0.5))
```

```
## [1] "numeric"
```

# Subsetting

We use square brackets to access specific elements of a vector. For the vector `codes` we defined above, we can access the second element using:

```
codes[2]
```

```
## canada
##    124
```

You can get more than one entry by using a multi-entry vector as an index:

```
codes[c(1,3)]
```

```
## italy egypt
##   380   818
```

# Subsetting

The sequences defined above are particularly useful if we want to access, say, the first two elements:

```r
codes[1:2]
```

```
##  italy canada
##    380    124
```

# Subsetting

If the elements have names, we can also access the entries using these names. Below are two examples.

```
codes["canada"]
```

```
## canada
##    124
```

```
codes[c("egypt","italy")]
```

```
## egypt italy
##   818   380
```

# Coercion

In general, **coercion** is an attempt by R to be flexible with data types. When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error. This can also lead to confusion.

Failing to understand **coercion** can drive programmers crazy in R since it behaves quite differently from most other languages in this regard. Let's learn about it with some examples.

# Coercion

We said that vectors must be all of the same type. So if we try to combine, say, numbers and characters, you might expect an error:

```
x <- c(1, "canada", 3)
```

But we don't get one, not even a warning! What happened? Look at `x` and its class:

```
x
```

```
## [1] "1"      "canada" "3"
```

```
class(x)
```

```
## [1] "character"
```

# Coercion

R **coerced** the data into characters. It guessed that because you put a character string in the vector, you meant the 1 and 3 to actually be character strings "1" and "3". The fact that not even a warning is issued is an example of how coercion can cause many unnoticed errors in R.

R also offers functions to change from one type to another. For example, you can turn numbers into characters with:

```
x <- 1:5
y <- as.character(x)
y
```

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

# Coercion

You can turn it back with `as.numeric`:

```
as.numeric(y)
```

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

This function is actually quite useful since datasets that include numbers as character strings are common.

# Not Availables (NA)

When a function tries to coerce one type to another and encounters an impossible case, it usually gives us a warning and turns the entry into a special value called an `NA` for "not available". For example:

```
x <- c("1", "b", "3")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1 NA  3
```

R does not have any guesses for what number you want when you type `b`, so it does not try. As a data scientist you will encounter the `NA`s often as they are generally used for missing data, a common problem in real-world datasets.

# 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   32
## [16]   36   38   53   63   65   67   84   93   93   97   97   99  111  116  118
## [31]  120  135  142  207  219  232  246  250  286  293  310  321  351  364  376
## [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"   "California"
## [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" "NE"
## [16] "OR" "DE" "MN" "KS" "CO" "NM" "NV" "AR" "WA" "CT" "WI" "DC" "OK" "KY" "MA"
## [31] "MS" "AL" "IN" "SC" "TN" "AZ" "NJ" "VA" "NC" "MD" "OH" "MO" "LA" "IL" "GA"
## [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.

# 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] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 185.42 170.18 177
```

# 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:

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

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

# 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):

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

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

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

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

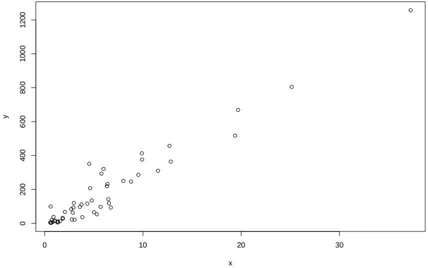
# Basic Plots

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:

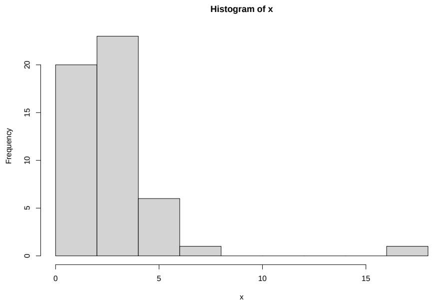
```
with(murders, plot(population, total))
```

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:

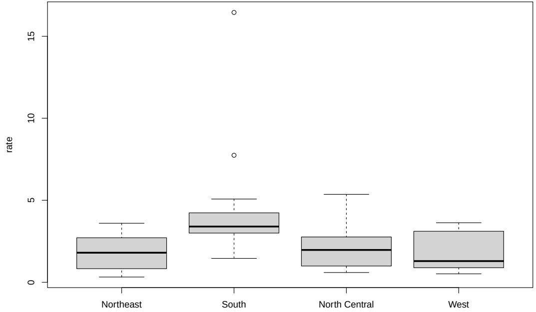
```
murders$state[which.max(x)]
```

```
## [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:

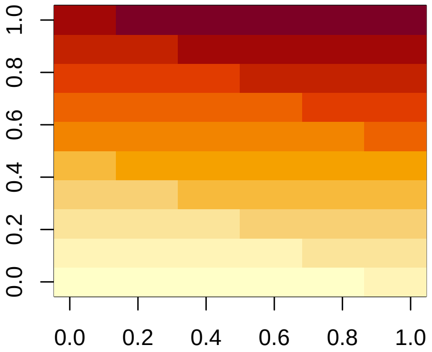
```
murders$rate <- with(murders, total / population * 100000)
boxplot(rate~region, data = murders)
```



# 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)
```



# Session Info

```r
sessionInfo()
```

```
## R version 4.4.2 (2024-10-31)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3.12.0
##
## 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_0.8.0      countrycode_1.6.0 ggflags_0.0.1     lubridate_1.9.4
##  [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_3.5.1
## [13] tidyverse_2.0.0
##
## loaded via a namespace (and not attached):
##  [1] generics_0.1.3     xml2_1.3.6         stringi_1.8.4      hms_1.1.3
##  [5] digest_0.6.37      magrittr_2.0.3     evaluate_1.0.0     grid_4.4.2
```