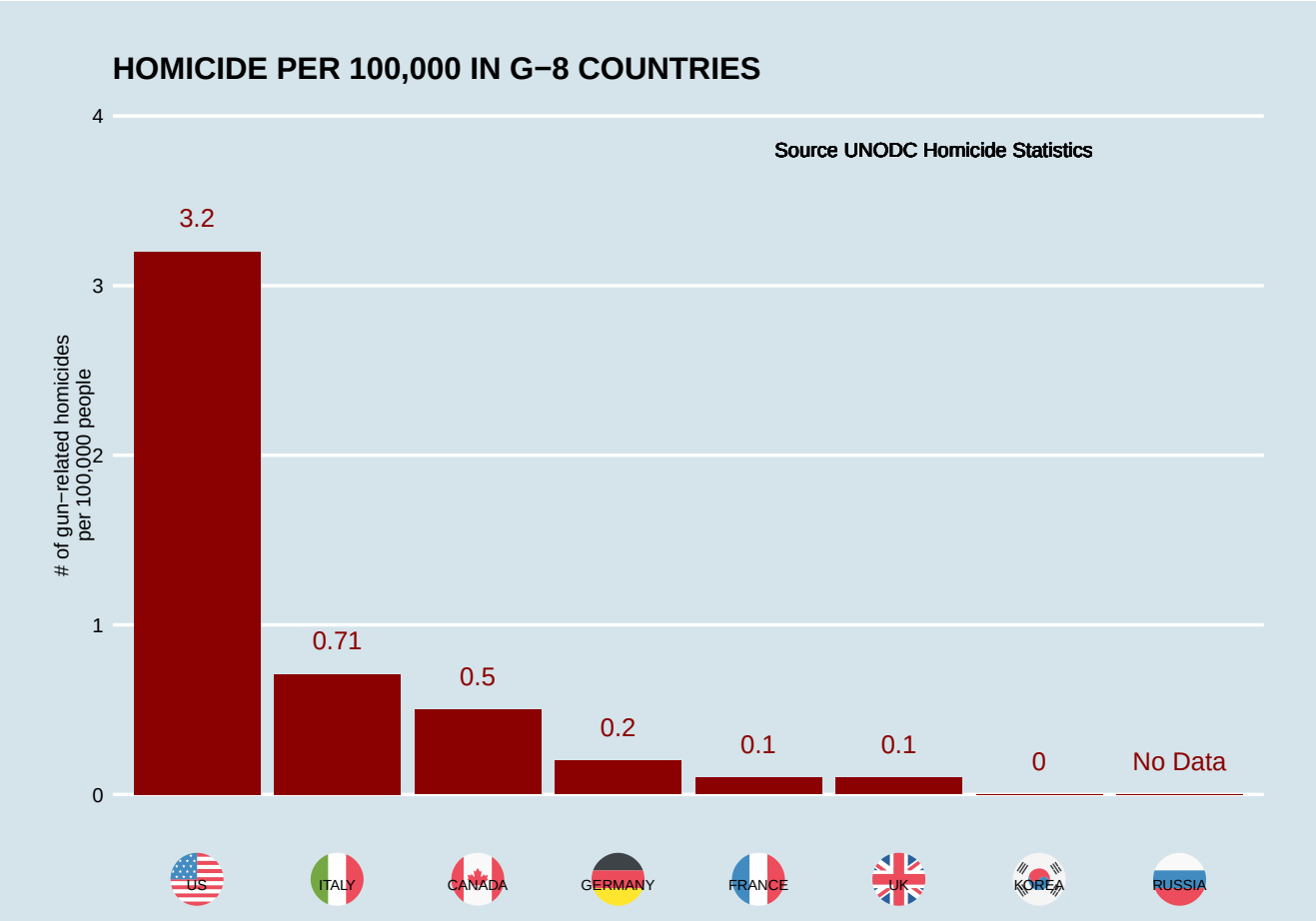
# 02 R basics

Soyoung Park

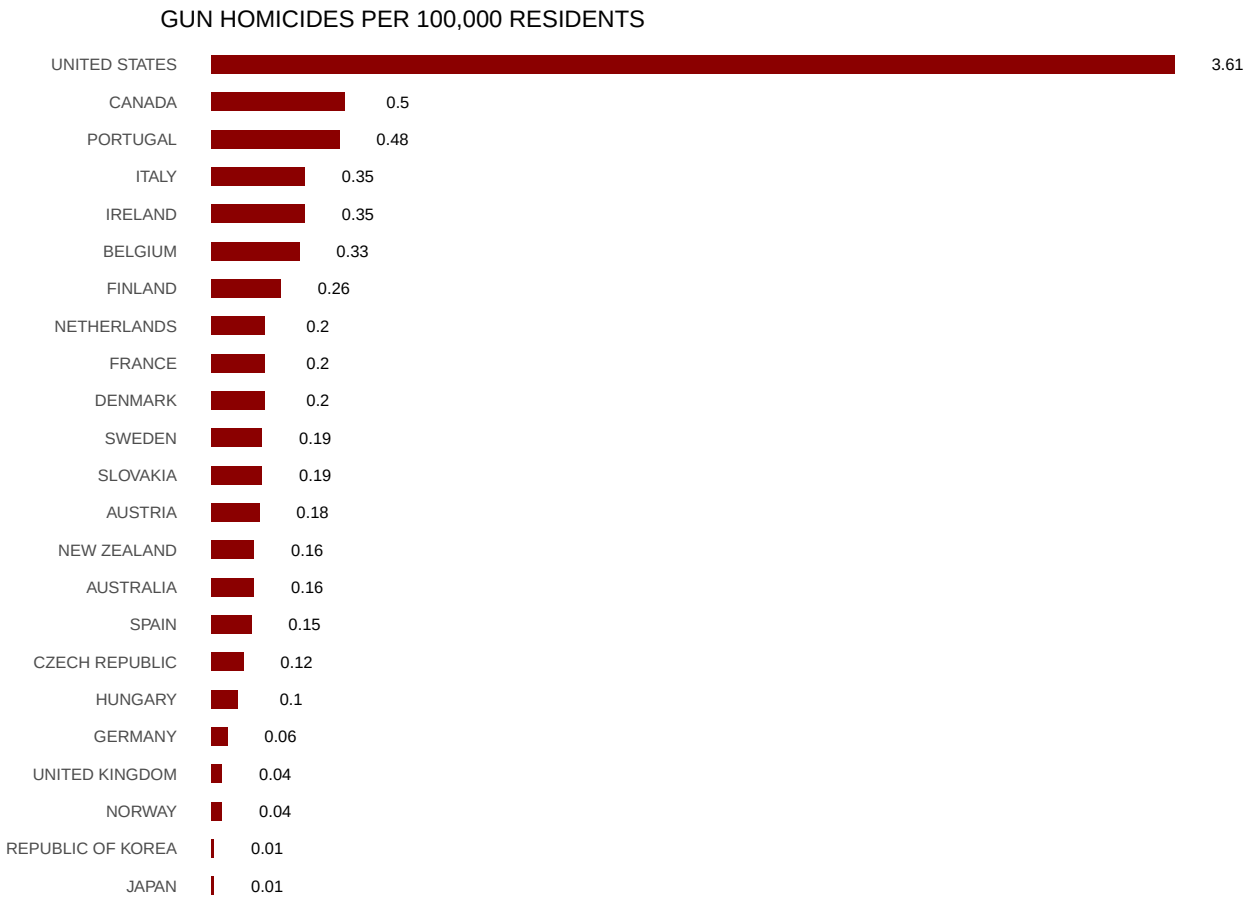
Pusan National University
Department of Statistics

# Case study: US Gun Murders



# Case study: US Gun Murders

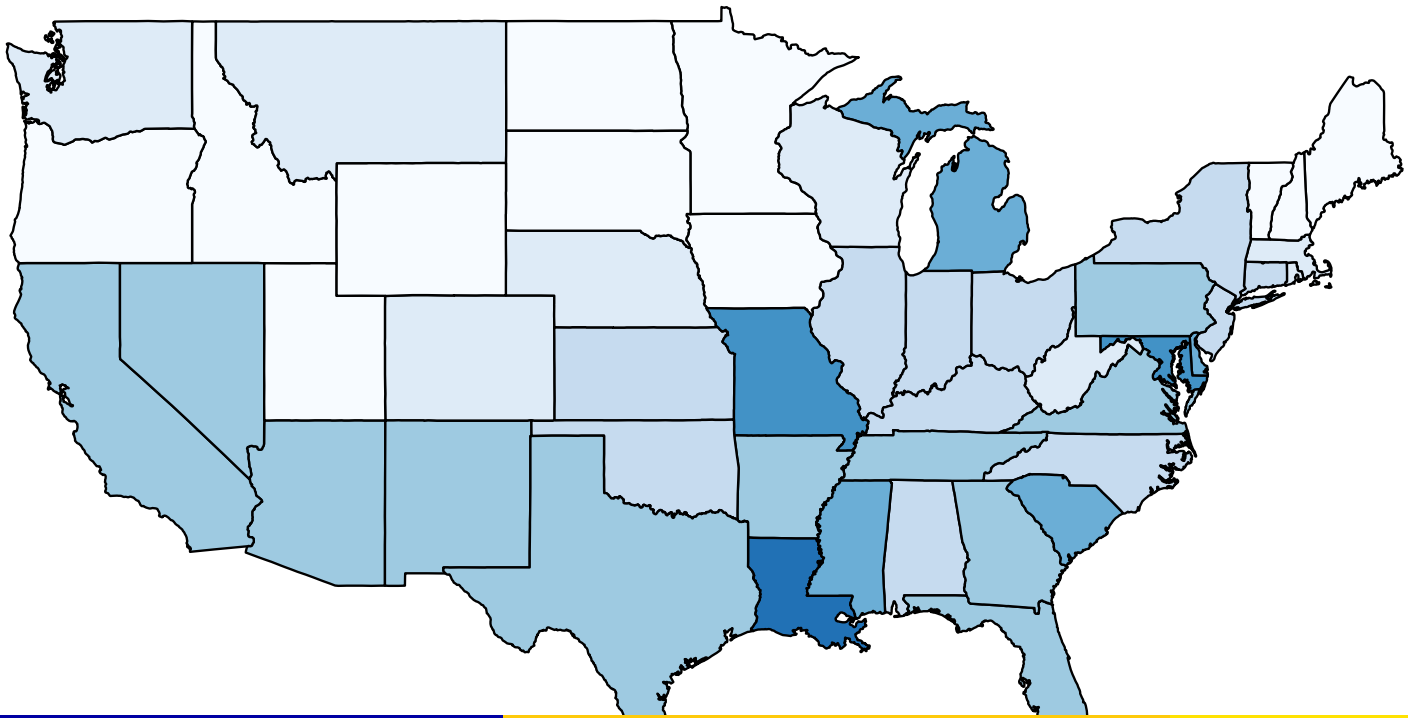
Or even worse, this version from everytown.org:



# Case study: US Gun Murders

But then you remember that the US is a large and diverse country with 50 very different states as well as the District of Columbia (DC).

```
## Warning: package 'dslabs' was built under R version 4.0.5
```



# The very basics - objects

To solve several 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 of course change depending on the values of $a$, $b$, and $c$.

# Objects

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:

```
a
```

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

```
b
```

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

```
c
```

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

# Objects

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

```
print(a);print(b);print(c)
```

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

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

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

```
a<-3
b<-6
c<--10

sqrt(2)
```

```
## [1] 1.414214
```

# The workspace

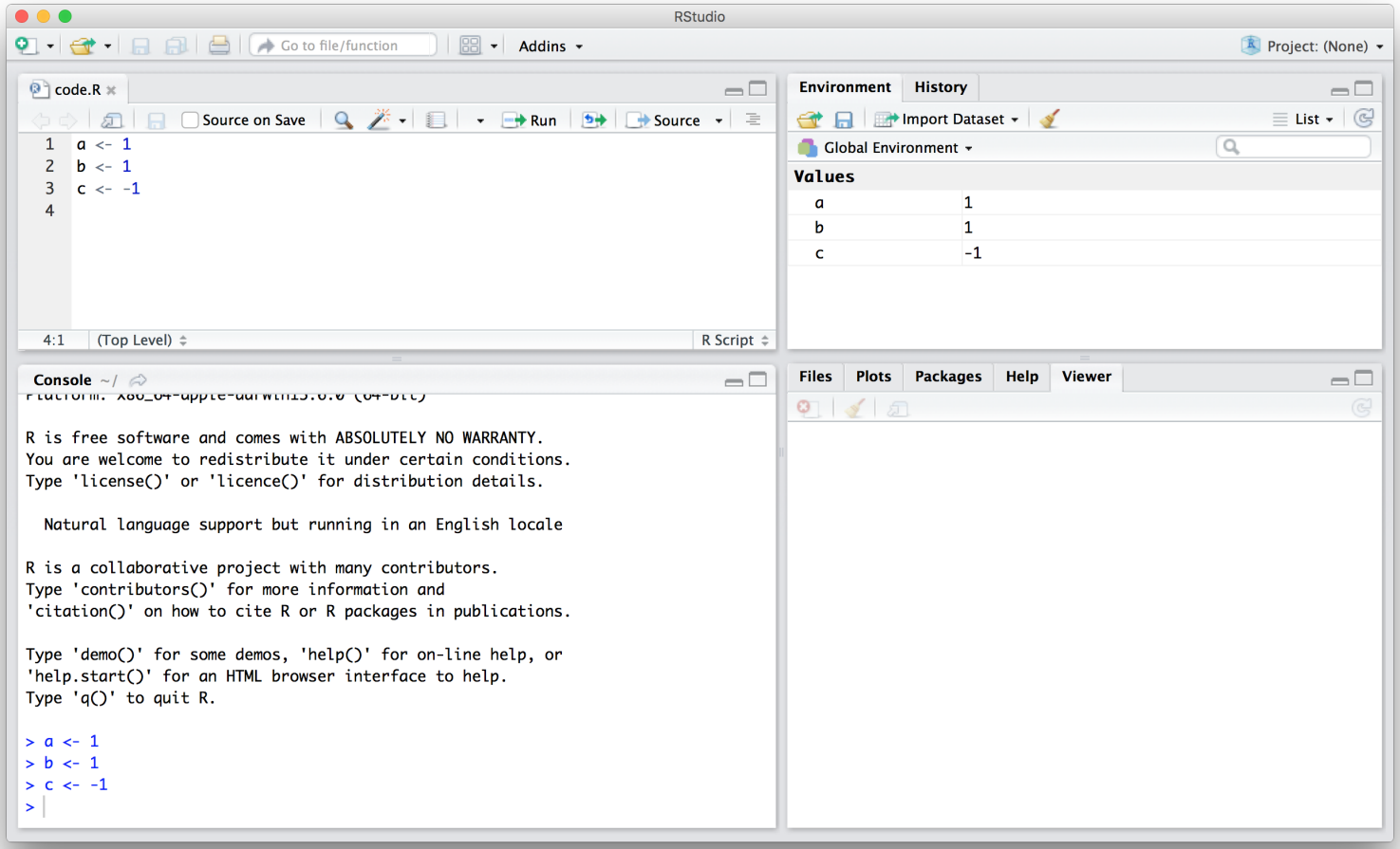
As we define objects in the console, we are actually changing the *workspace*. You can see all the variables saved in your workspace by typing:

```
ls()
```

```
##  [1] "a"                "b"                    "c"
##  [5] "img_path"         "mac_screenshots" "murders"
##  [9] "x1"               "x2"
```

# 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`.

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] 1.081666
```

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

```
## [1] -3.081666
```

# Functions

R includes several **predefined** functions and most of the analysis pipelines we construct make extensive use of these. Functions that we have used:

- `install.packages`
- `library`
- `ls`
- `sqrt` ... more

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, such as `ls()`.

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] 1.098612
```

# 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")
?log
```

The help page will show you what arguments the function is expecting.

# Functions

The base of the function `log` defaults to `base = exp(1)` making `log` the natural log by default.

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

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

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

# Functions

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

```
2 ^ 3
```

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

You can see the arithmetic operators by typing:

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

or

```
?"+"
?">"
```

# Variable names

We have used the letters `a`, `b`, and `c` as variable names, but variable names can be almost anything.

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

```
sol1 <- (-b + sqrt(b^2 - 4*a*c)) / (2*a)
sol2 <- (-b - sqrt(b^2 - 4*a*c)) / (2*a)
```

Caution: Don't name one of your variables `install.packages` (name of functions) by typing something like `install.packages <- 2`.

# Saving your workspace

- Values remain in the workspace until you end your session or erase them with the function `rm`.
- But workspaces also can be saved for later use.
- 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

- We recommend you assign the workspace a specific name. You can do this by using the function `save` or `save.image`.
- To load, use the function `load`. When saving a workspace, 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.

# Commenting your code

If a line of R code starts with the symbol #, it is not evaluated. For example, in the script above we could add:

```
## Code to compute solution to quadratic equation of the form
## 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)
```

# Exercises

1. What is the sum of the first 100 positive integers? The formula for the sum of integers 1 through $n$ is $n(n+1)/2$. Define $n = 100$ and then use R to compute the sum of 1 through 100 using the formula. What is the sum?

# Exercises

2. Now use the same formula to compute the sum of the integers from 1 through 1,000.

# Exercises

3. Look at the result of typing the following code into R:

```
n <- 1000
x <- seq(1, n)
sum(x)
```

Based on the result, what do you think the functions `seq` and `sum` do? You can use `help`.

a. `sum` creates a list of numbers and `seq` adds them up.
b. `seq` creates a list of numbers and `sum` adds them up.
c. `seq` creates a random list and `sum` computes the sum of 1 through 1,000.
d. `sum` always returns the same number.

# Exercises

4. In math and programming, we say that we evaluate a function when we replace the argument with a given number. So if we type `sqrt(4)`, we evaluate the `sqrt` function. In R, you can evaluate a function inside another function. The evaluations happen from the inside out. Use one line of code to compute the log, in base 10, of the square root of 100.

# Exercises

5. Which of the following will always return the numeric value stored in `x`? You can try out examples and use the help system if you want.

a. `log(10^x)`
b. `log10(x^10)`
c. `log(exp(x))`
d. `exp(log(x, base = 2))`

# Data types

Variables in R can be of different types. For example, we need to distinguish numbers from character strings and tables from simple lists of numbers. 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.

# type of variable

- `int` stands for integers.
- `dbl` stands for doubles, or real numbers.
- `chr` stands for character vectors, or strings.
- `dttm` stands for date-times (a date + a time).
- `lgl` stands for logical, vectors that contain only TRUE or FALSE.
- `fctr` stands for factors, which R uses to represent categorical variables with fixed possible values.
- `date` stands for dates.

# Vectors: numeric

The object `murders$population` is not one number but several. We call these types of objects *vectors*.

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

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

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: character

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: logical

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

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

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

```
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: relational

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

```
?Comparison

x<-1
y<-2

x<y #TRUE
x==y #FALSE
x!=y #TRUE
```

# Vectors

**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:

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

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

It is a ***factor***. Factors are useful for storing categorical data. We can see that there are only 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.

# Change the orders of the level

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.

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

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

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

# Data frames

Up to now, the variables we have defined are just one number. This 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.

You can access this dataset by loading the **dslabs** library and loading the `murders` dataset using the `data` function:

```
library(dslabs)
library(tidyverse)
data(murders)
class(murders)
```

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

# Examining a data

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
##  $ population: num  4779736 710231 6392017 2915918 3725395
##  $ total     : num  135 19 232 93 1257 ...
```

# Examining a data

A head of `murders`:

```
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
```

```
glimpse(murders)
```

```
## Rows: 51
## Columns: 5
## $ state      <chr> "Alabama", "Alaska", "Arizona", "Arkansa
```

# Examining a data

Reveal the names for each of the five variables stored in this dataframe, use `names`:

```
names(murders)
```

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

# The accessor: $

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
##  [9]   601723 19687653  9920000  1360301  1567582 12830632
## [17]  2853118  4339367  4533372  1328361  5773552  6547629
## [25]  2967297  5988927   989415  1826341  2700551  1316470
## [33] 19378102  9535483   672591 11536504  3751351  3831074
## [41]  4625364   814180  6346105 25145561  2763885   625741
## [49]  1852994  5686986   563626
```

```
#murders$p  ## works but not recommended
```

# 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")
```

This list includes a character, a number, a vector with five numbers, and another character.

```
#record
class(record)
```

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

# Lists

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

```
record$student_id
```

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

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

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

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

```
record[[2]]
```

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

You should get used to the fact that in R, there are often several ways to

# Lists

You might also encounter lists without variable names.

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

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

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]]
```

# 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.
- Yet matrices have a major advantage over data frames: we can perform matrix algebra operations, a powerful type of mathematical technique.

# 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
```

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

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

# 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.

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

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

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

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

```
##   V1 V2 V3
## 1  1  5  9
```

# Matrices

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

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

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

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

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

# Exercises

1. Load the US murders dataset.

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

Use the function `str` to examine the structure of the `murders` object. Which of the following best describes the variables represented in this data frame?

a. The 51 states.
b. The murder rates for all 50 states and DC.
c. The state name, the abbreviation of the state name, the state's region, and the state's population and total number of murders for 2010.
d. `str` shows no relevant information.

# Exercises

2. What are the column names used by the data frame for these five variables?

3. Use the accessor $ to extract the state abbreviations and assign them to the object a. What is the class of this object?

# Exercises

4. Now use the square brackets to extract the state abbreviations and assign them to the object `b`. Use the `identical` function to determine if `a` and `b` are the same.

# Exercises

5. We saw that the `region` column stores a factor. You can corroborate this by typing:

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

With one line of code, use the function `levels` and `length` to determine the number of regions defined by this dataset.

# Exercises

6. The function `table` takes a vector and returns the frequency of each element. You can quickly see how many states are in each region by applying this function. Use this function in one line of code to create a table of states per region.

# Vectors

In R, the most basic objects available to store data are *vectors*. Complex datasets can usually be broken down into components that are 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
```

# 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")
```

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* `.

# 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
```

The object `codes` continues to be a numeric vector:

```
class(codes)
```

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

but with names:

# 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
```

```
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. First and the second argument are defined the start and end, the third one is how much to jump by:

```
#?seq()
seq(1, 10)
```

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

```
seq(1, 10, 2) # 1, 3, 5, 7, 9
```

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

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

```
1:10
```

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

# 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**.

```
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:

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

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

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")]
```

# Coercion

In general, *coercion* is an attempt by R to be flexible with data types.

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
class(x)
```

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

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".

# Coercion

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"
```

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)
```

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.

# Exercises

1. Use the function `c` to create a vector with the average high temperatures in January for Beijing, Lagos, Paris, Rio de Janeiro, San Juan, and Toronto, which are 35, 88, 42, 84, 81, and 30 degrees Fahrenheit. Call the object `temp`.

2. Now create a vector with the city names and call the object `city`.

3. Use the `names` function and the objects defined in the previous exercises to associate the temperature data with its corresponding city.

# Exercises

4. Use the `[` and `:` operators to access the temperature of the first three cities on the list.

5. Use the `[` operator to access the temperature of Paris and San Juan.

6. Use the `:` operator to create a sequence of numbers $12, 13, 14, \ldots, 73$.

# Exercises

7. Create a vector containing all the positive odd numbers smaller than 100.

8. Create a vector of numbers that starts at 6, does not pass 55, and adds numbers in increments of 4/7: 6, 6 + 4/7, 6 + 8/7, and so on. How many numbers does the list have? Hint: use `seq` and `length`.

9. What is the class of the following object `a <- seq(1, 10, 0.5)`?

# Exercises

10. What is the class of the following object `a <- seq(1, 10)`?

11. The class of `class(a<-1)` is numeric, not integer. R defaults to numeric and to force an integer, you need to add the letter `L`. Confirm that the class of `1L` is integer.

12. Define the following vector:

```
x <- c("1", "3", "5")
```

and coerce it to get integers.

# Sorting: `sort`

Say we want to rank the states from least to most gun murders. The function `sort` sorts a vector in increasing order.

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

```
##  [1]    2    4    5    5    7    8   11   12   12   16   19
## [16]   36   38   53   63   65   67   84   93   93   97   97
## [31]  120  135  142  207  219  232  246  250  286  293  310
## [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.

# Sorting:order

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.

```
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:order

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 and so on.

# Sorting:order

How does this help us order the states by murders?

For example, these two vectors containing state names and abbreviations, respectively, are matched by their order:

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

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

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

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

This means we can **order the state names by their total murders**.

# Sorting:order

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"
## [16] "OR" "DE" "MN" "KS" "CO" "NM" "NV" "AR" "WA" "CT" "WI"
## [31] "MS" "AL" "IN" "SC" "TN" "AZ" "NJ" "VA" "NC" "MD" "OH"
## [46] "MI" "PA" "NY" "FL" "TX" "CA"
```

According to the above, California had the most murders.

# max and which.max

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.

# `max` and `which.max`

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`.

# 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: summary

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

```
##   original sort order rank
## 1       31    4     2    3
## 2        4   15     3    1
## 3       15   31     1    2
## 4       92   65     5    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 element-wise. 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
```

```
## [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

For these exercises we will use the US murders dataset. Make sure you load it prior to starting.

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

1. Use the $ operator to access the population size data and store it as the object `pop`. Then use the `sort` function to redefine `pop` so that it is sorted. Finally, use the [ operator to report the smallest population size.

2. Now instead of the smallest population size, find the index of the entry with the smallest population size. Hint: use `order` instead of `sort`.

# Exercises

3. We can actually perform the same operation as in the previous exercise using the function `which.min`. Write one line of code that does this.

4. Now we know how small the smallest state is and we know which row represents it. Which state is it? Define a variable `states` to be the state names from the `murders` data frame. Report the name of the state with the smallest population.

# Exercises

5. You can create a data frame using the `data.frame` function. Here is a quick example:

```
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro",
          "San Juan", "Toronto")
city_temps <- data.frame(name = city, temperature = temp)
```

Use the `rank` function to determine the population rank of each state from smallest population size to biggest. Save these ranks in an object called `ranks`, then create a data frame with the state name and its rank. Call the data frame `my_df`.

# Exercises

6. Repeat the previous exercise, but this time order `my_df` so that the states are ordered from least populous to most populous. Hint: create an object `ind` that stores the indexes needed to order the population values. Then use the bracket operator `[` to re-order each column in the data frame.

7. The `na_example` vector represents a series of counts. You can quickly examine the object using:

```
data("na_example")
str(na_example)
```

```
##  int [1:1000] 2 1 3 2 1 3 1 4 3 2 ...
```

# Exercises

However, when we compute the average with the function `mean`, we obtain an `NA`:

```
mean(na_example)
```

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

The `is.na` function returns a logical vector that tells us which entries are `NA`. Assign this logical vector to an object called `ind` and determine how many `NA`s does `na_example` have.

# Exercises

8. Now compute the average again, but only for the entries that are not `NA`. Hint: remember the `!` operator.

# Vector arithmetics

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"
```

What we really should be computing is the **murders per capita**.

# Vector arithmetics

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 or subtract 69 inches (the average):

```
inches * 2.54 # to centimeters
```

```
##  [1] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 185.4
```

```
inches - 69   # how much taller or shorter than the average 6
```

```
##  [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 mathematical operations, such as `-`, `*` and `/`.

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

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"
## [16] "WV" "RI" "WI" "NE" "MA" "IN" "KS" "NY" "KY" "AK" "OH"
## [31] "OK" "NC" "NV" "VA" "AR" "TX" "NM" "CA" "FL" "TN" "PA"
## [46] "DE" "SC" "MD" "MO" "LA" "DC"
```

# Exercises

1. Previously we created this data frame:

```
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro",
          "San Juan", "Toronto")
city_temps <- data.frame(name = city, temperature = temp)
```

Remake the data frame using the code above, but add a line that converts the temperature from Fahrenheit to Celsius. The conversion is $C = \frac{5}{9} \times (F - 32)$.

# Exercises

2. What is the following sum $1 + 1/2^2 + 1/3^2 + \ldots 1/100^2$? Hint: thanks to Euler, we know it should be close to $\pi^2/6$.

3. Compute the per 100,000 murder rate for each state and store it in the object `murder_rate`. Then compute the average murder rate for the US using the function `mean`. What is the average?

# Indexing

R provides a powerful and convenient way of indexing vectors. We can, for example, subset a vector based on properties of another vector. In this section, we continue working with our US murders example, which we can load like this:

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

# Subsetting with logicals

We have now calculated the murder rate using:

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

- 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.

# Subsetting with logicals

The following is an example related to the question above:

```
ind <- murder_rate < 0.71
#ind <- murder_rate <= 0.71 #if a value is less or equal
```

Note that we get back a logical vector with `TRUE` for each entry smaller than or equal to 0.71.

# Subsetting with logicals

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
## [5] "Vermont"
```

In order to count how many are TRUE, the function `sum` returns the sum of the entries of a vector and logical vectors with `TRUE` coded as 1 and `FALSE` as 0.

```
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"
```

# 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
```

# 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
```

# %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 the book.

# Notice

**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

Start by loading the library and data.

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

1. Compute the per 100,000 murder rate for each state and store it in an object called `murder_rate`. Then use logical operators to create a logical vector named `low` that tells us which entries of `murder_rate` are lower than 1.

2. Now use the results from the previous exercise and the function `which` to determine the indices of `murder_rate` associated with values lower than 1.

3. Use the results from the previous exercise to report the names of the states with murder rates lower than 1.

# Exercises

4. Now extend the code from exercises 2 and 3 to report the states in the Northeast with murder rates lower than 1. Hint: use the previously defined logical vector `low` and the logical operator `&`.

5. In a previous exercise we computed the murder rate for each state and the average of these numbers. How many states are below the average?

6. Use the match function to identify the states with abbreviations AK, MI, and IA. Hint: start by defining an index of the entries of `murders$abb` that match the three abbreviations, then use the `[` operator to extract the states.

# Exercises

7. Use the `%in%` operator to create a logical vector that answers the question: which of the following are actual abbreviations: MA, ME, MI, MO, MU?

8. Extend the code you used in exercise 7 to report the one entry that is **not** an actual abbreviation. Hint: use the `!` operator, which turns `FALSE` into `TRUE` and vice versa, then `which` to obtain an index.

# Basic plots

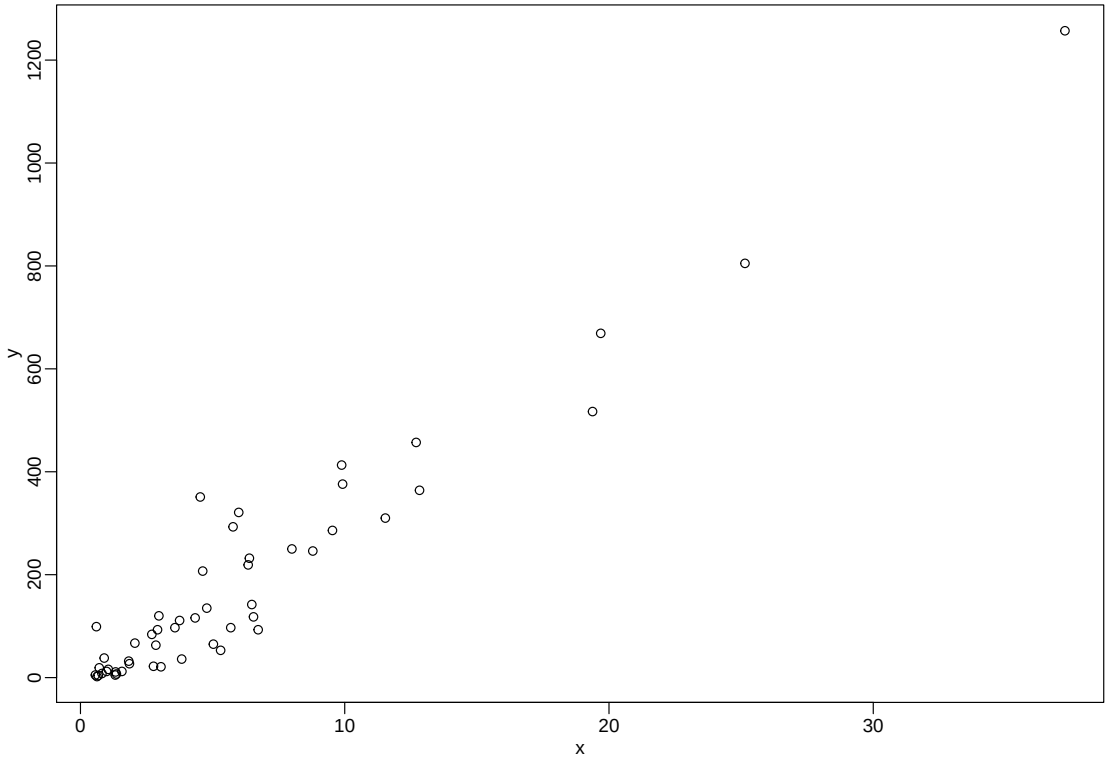
We briefly describe some of the functions that are available in a basic R installation.

# 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)
```

# plot



# 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.

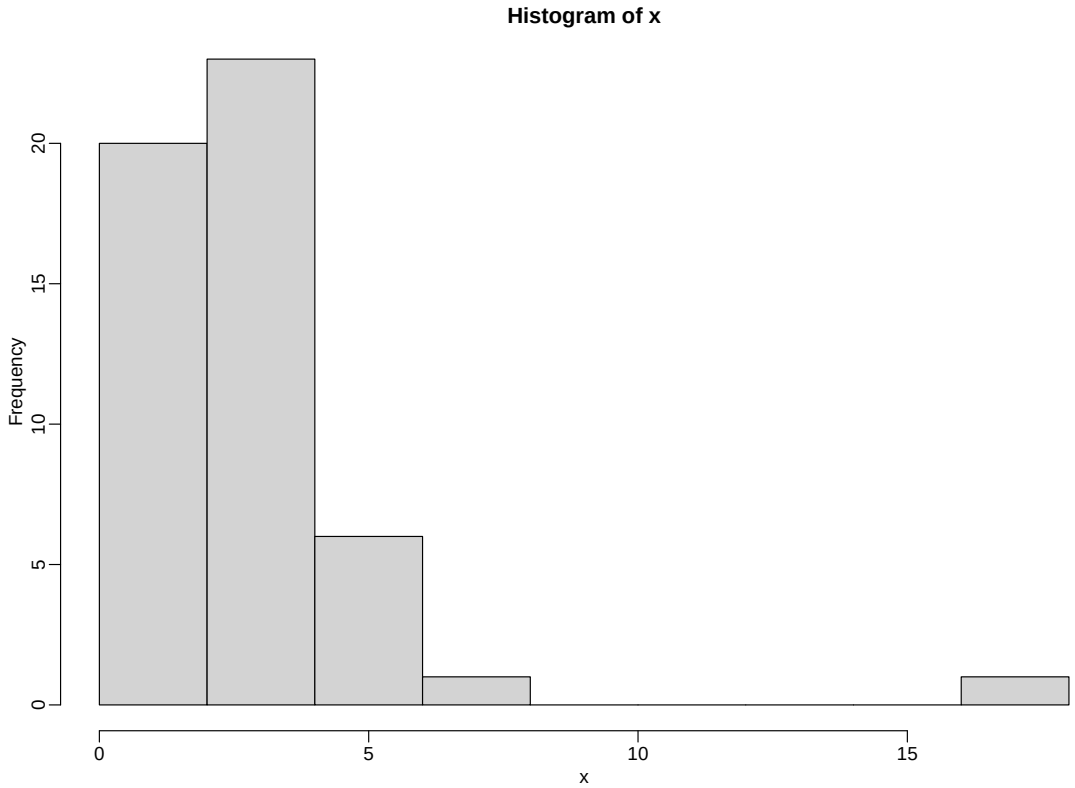
# hist

We will describe histograms as they relate to distributions in the Data Visualization part of the book.

We can make a histogram of our murder rates by simply typing:

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

# hist



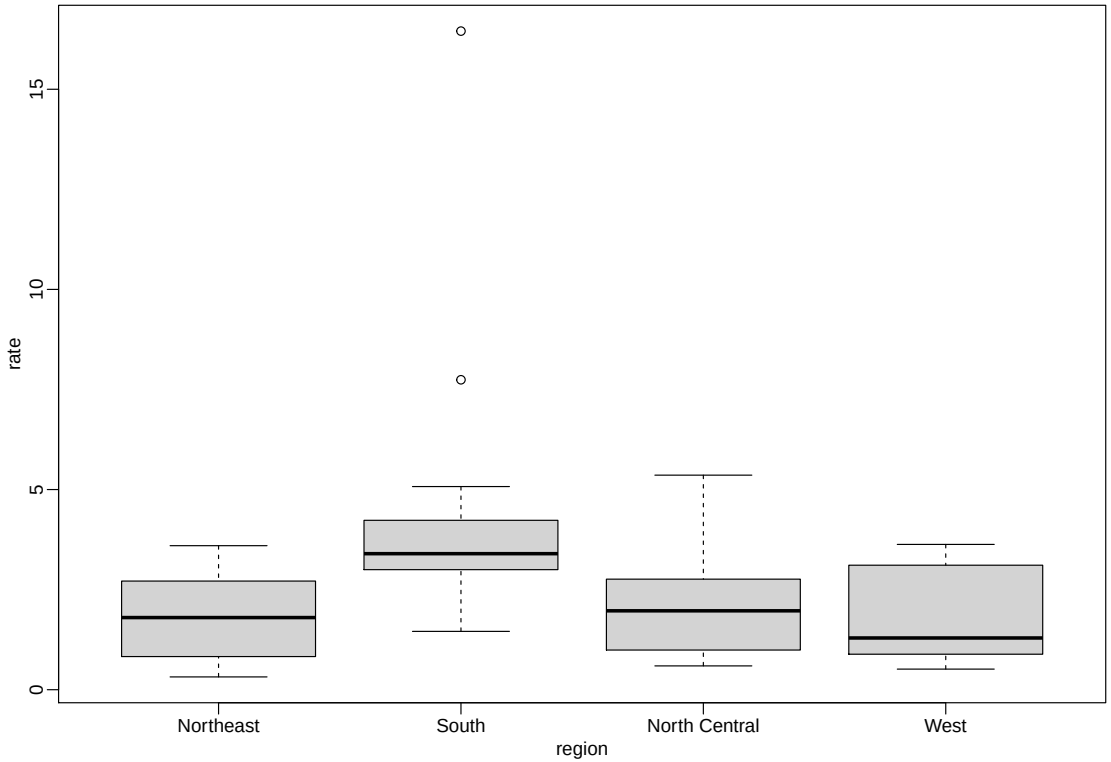

We can see that there is a wide range of values with most of them between 2 and 3 and one very extreme case with a murder rate of more than 15:

# boxplot

Boxplots will also be described in the Data Visualization part of the book.

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

# boxplot



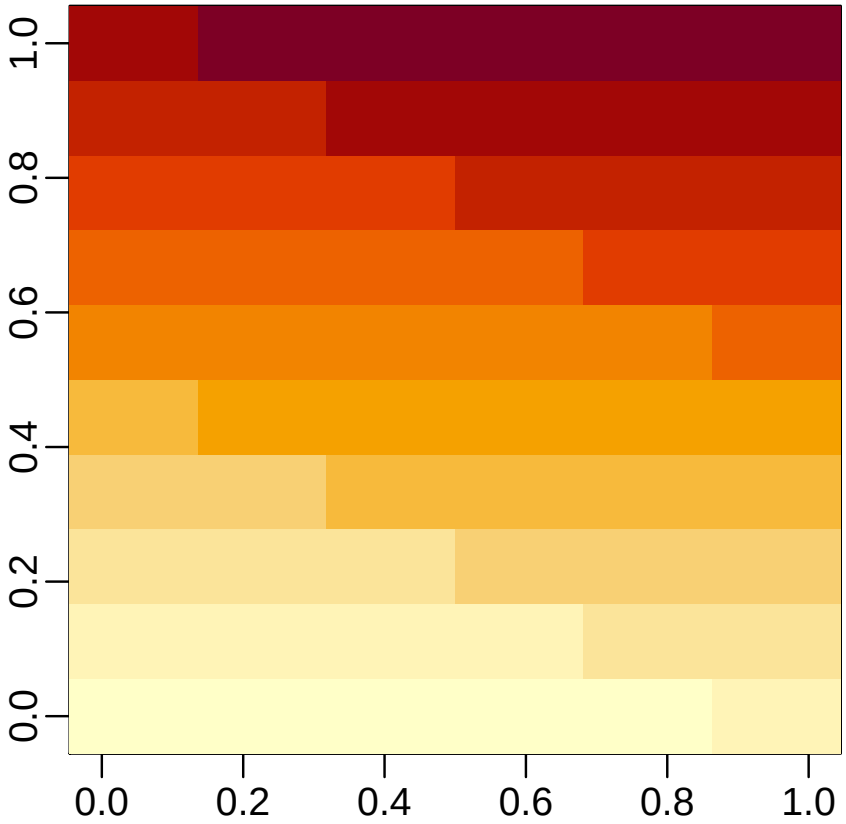

We can see that the South has higher murder rates than the other three regions.

# 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)
```

# image



# Exercises

1. We made a plot of total murders versus population and noted a strong relationship. Not surprisingly, states with larger populations had more murders.

```
library(dslabs)
data(murders)
population_in_millions <- murders$population/10^6
total_gun_murders <- murders$total
plot(population_in_millions, total_gun_murders)
```

Keep in mind that many states have populations below 5 million and are bunched up. We may gain further insights from making this plot in the log scale. Transform the variables using the `log10` transformation and then plot them.

# Exercises

2. Create a histogram of the state populations.
3. Generate boxplots of the state populations by region.