Lecture 1: R Basics

Nick Hagerty*

ECNS 491/560 Fall 2022

Montana State University

*Parts of these slides are adapted from “Introduction to Data Science” by Rafael A. Irizarry, used under CC BY-NC-SA 4.0.

Table of contents

1. About R

2. Operators

3. Objects and functions

4. Data frames

5. Vectors

6. Miscellaneous basics

About R

Why are we using R in this course?

- It’s free and open source
- It’s widely used in industry
- It’s widely used in academic research
- It has a large and active user community

Compared with Stata:

- More of a true programming language
- Steeper learning curve (takes more to get started, but ultimately more powerful)
- Many advantages I'll point out throughout the course

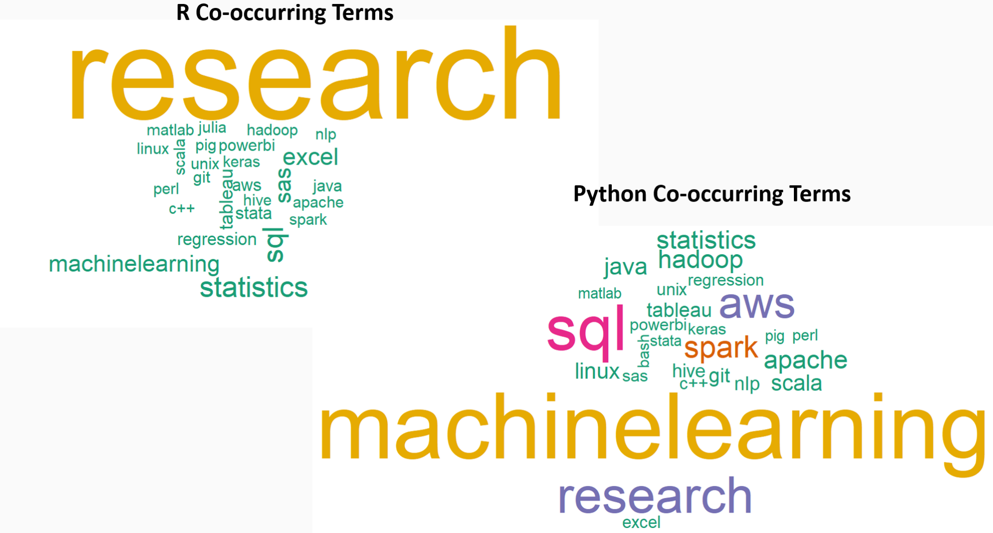
R vs. Python

R:

- Built for statistics and data analysis
- Better at econometrics and data visualization

Python:

- Built for general-purpose programming and software development
- Better at machine learning



R Co-occurring Terms

Python Co-occurring Terms

Image by Alex daSilva (source) is not included under the CC license.

R vs. Python

R:

- Built for statistics and data analysis
- Better at econometrics and data visualization

Python:

- Built for general-purpose programming and software development
- Better at machine learning

Most economists use either Stata or R

Many data scientists in industry use both R and Python

Rising competitor to both: Julia

R is a means, not an end

- The goals of this course are **platform-agnostic**
    - It’s not about the syntax of specific packages
    - It’s about the concepts, logic, and thought processes underlying what we're doing and why
- Your eventual goal: **Use the right tool for the job**
- Personally, I probably still have a bit more expertise in Stata than R
- Many of you will know more than me about some of the things we’re learning about
    - Please speak up and share!

R and RStudio

- R is like the car's engine

- RStudio is the dashboard

Getting to know RStudio

1. **Tour of panes:** Console, environment, scripts, other stuff
2. **Try out the console**
   - Use it as a calculator
   - Access previous commands
3. **Try a new script and save it**
4. **Set global options (Tools -> Options)**
   - Uncheck "Restore .RData into workspace at start"
   - Set "Save workspace to .RData on exit" to "Never"
5. **Keyboard shortcuts**

Time for some live coding

Open a **new R script.**

As we go through examples, **retype everything yourself and run it line by line** (ctl+enter). You'll learn more this way.

(Feel free to try out slight tweaks along the way, too.)

Operators

Basic arithmetic

You can use R like a fancy graphing calculator:

```
1 + 2 # Addition
```

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

```
6 - 7 # Subtraction
```

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

```
5 / 2 # Division
```

```
## [1] 2.5
```

```
2 ^ 3 # Exponentiation
```

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

```
2 + 4 * 1 ^ 3 # Standard order of operations
```

```
## [1] 6
```

Logical evaluation

Logical operators follow standard programming conventions:

```
1 > 2
```

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

```
1 > 2 & 1 > 0.5 # The "&" means "and"
```

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

```
1 > 2 | 1 > 0.5 # The "|" means "or"
```

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

Negation:

```
!(1 > 2)
```

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

Commenting

R ignores the rest of a line after a `#`. So you can write notes to yourself about what your code is doing.

```r
# Test whether 4 is greater than 3
4 > 3
```

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

Widely accepted conventions:

- Put the comment **before** the code it refers to.
- Use present tense.

Evaluation

This doesn't work, because `=` is reserved for assignment:

```
1 = 1
```

```
## Error in 1 = 1: invalid (do_set) left-hand side to assignment
```

Instead, use **==**:

```
1 == 1
```

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

For "not equal", use **!=**:

```
1 != 2   # This looks weird because of the font
```

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

Note: **Read the error message!** What should you do if you don't understand it?

Objects and functions

Objects

We can store values for later by assigning them to **objects.**

```
bill = 18.45
percentage = 0.2
```

Instead of **=**, you can use **<-** (and many people do):

```
bill ← 18.45  # this font turns "<" and "-" into a symbol
percentage ← 0.2
```

In this course, I will use `=` for assignment. You can use either one, but be consistent.

Objects

To see the value of an object, just type its name:

```
bill
```

```
## [1] 18.45
```

Notice that `bill` and `percentage` are now listed in your Environment pane.

Now, we can calculate the tip:

```
bill * percentage
```

```
## [1] 3.69
```

Assign a new value to `bill` and recalculate the tip:

```
bill = 90
bill * percentage
```

```
## [1] 18
```

Challenge

Try on your own, and compare your solution with a neighbor:

Calculate the sum of the first 100 positive integers.

Hint: The formula for the sum of integers 1 through n is $n(n+1)/2$.

Using functions

Doing anything more complicated than arithmetic requires **functions.**

```
log(50)
```

```
## [1] 3.912023
```

To find out what **arguments** a function takes, look up its help file.

```
?log
```

Some arguments are required, some are optional. You can see that `base` is optional because it has a default value: `exp(1)`.

If you type the arguments in the expected order, you don't need to use argument names:

```
log(50, 10)
```

```
## [1] 1.69897
```

Using functions

But using argument names can help improve clarity:

```
log(50, base = 10)
```

```
## [1] 1.69897
```

If you name all the arguments, you can put them in any order:

```
log(base = 10, x = 50)
```

```
## [1] 1.69897
```

We can use objects as arguments, or nest functions:

```
log(bill)
```

```
## [1] 4.49981
```

```
log(exp(50))
```

```
## [1] 50
```

Data types

There are many different types of objects:

- vectors (numeric, character, logical, integer)
- matrices
- data frames
- lists
- functions

To know what type of object you have, use `class`:

```r
a = 2
class(a)
```

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

```r
class("a")
```

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

```r
class(TRUE)
```

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

Data frames

Packages

Many of the most useful functions of R come from add-on **packages.**

To install the package called `dslabs`, type:

```r
install.packages("dslabs")
```

You only need to install a package on your computer once. But you still need to load it each time you open RStudio:

```r
library(dslabs)
```

Load the dataset `murders` from this package:

```r
data(murders)
```

Data frames

A data frame is like a table. Each row is an observation and each column is a variable.

```
class(murders)
```

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

To learn more about an data frame, you can:

(1) Examine its **str**ucture with `str`:

```
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 2 ...
##  $ population: num  4779736 710231 6392017 2915918 37253956 ...
##  $ total     : num  135 19 232 93 1257 ...
```

Data frames

(2) Display some summary statistics with `summary`:

```
summary(murders)
```

```
##     state               abb                        region     population
##  Length:51          Length:51          Northeast    : 9   Min.   :  563626
##  Class :character   Class :character   South        :17   1st Qu.: 1696962
##  Mode  :character   Mode  :character   North Central:12   Median : 4339367
##                                        West         :13   Mean   : 6075769
##                                                           3rd Qu.: 6636084
##                                                           Max.   :37253956
##      total
##  Min.   :   2.0
##  1st Qu.:  24.5
##  Median :  97.0
##  Mean   : 184.4
##  3rd Qu.: 268.0
##  Max.   :1257.0
```

Data frames

(3) Show the first few rows with `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
```

(4) Directly inspect it with `View` (or just click on it in your Environment pane)

```
View(murders)
```

The accessor ($)

To refer to individual variables (columns) in this data frame, we can use `$`:

```
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 object `murders$population` is a **vector**, a set of numbers.

How many entries (rows) does it have?

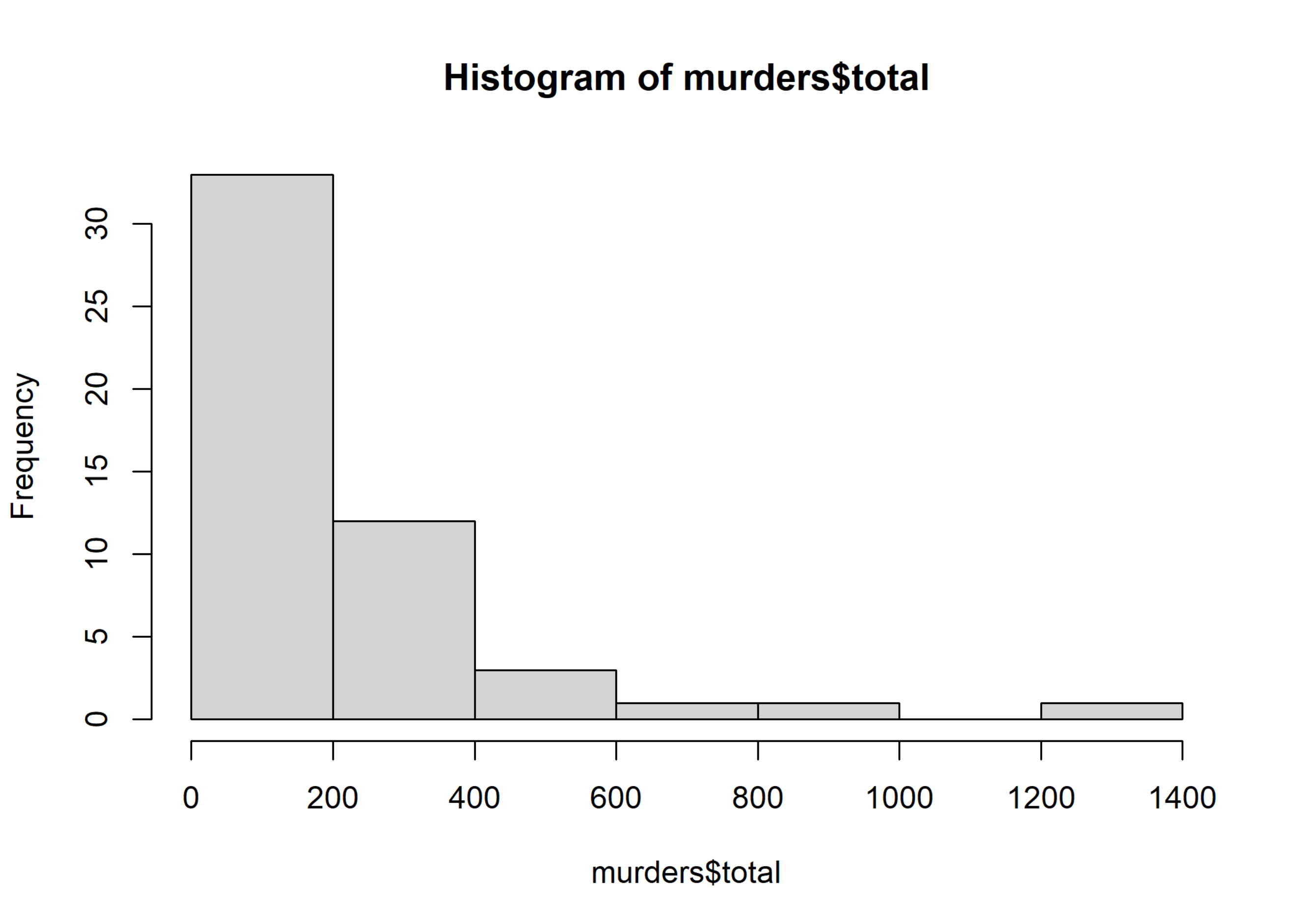
```
length(murders$population)
```

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

Basic plots

Make a quick **histogram:**

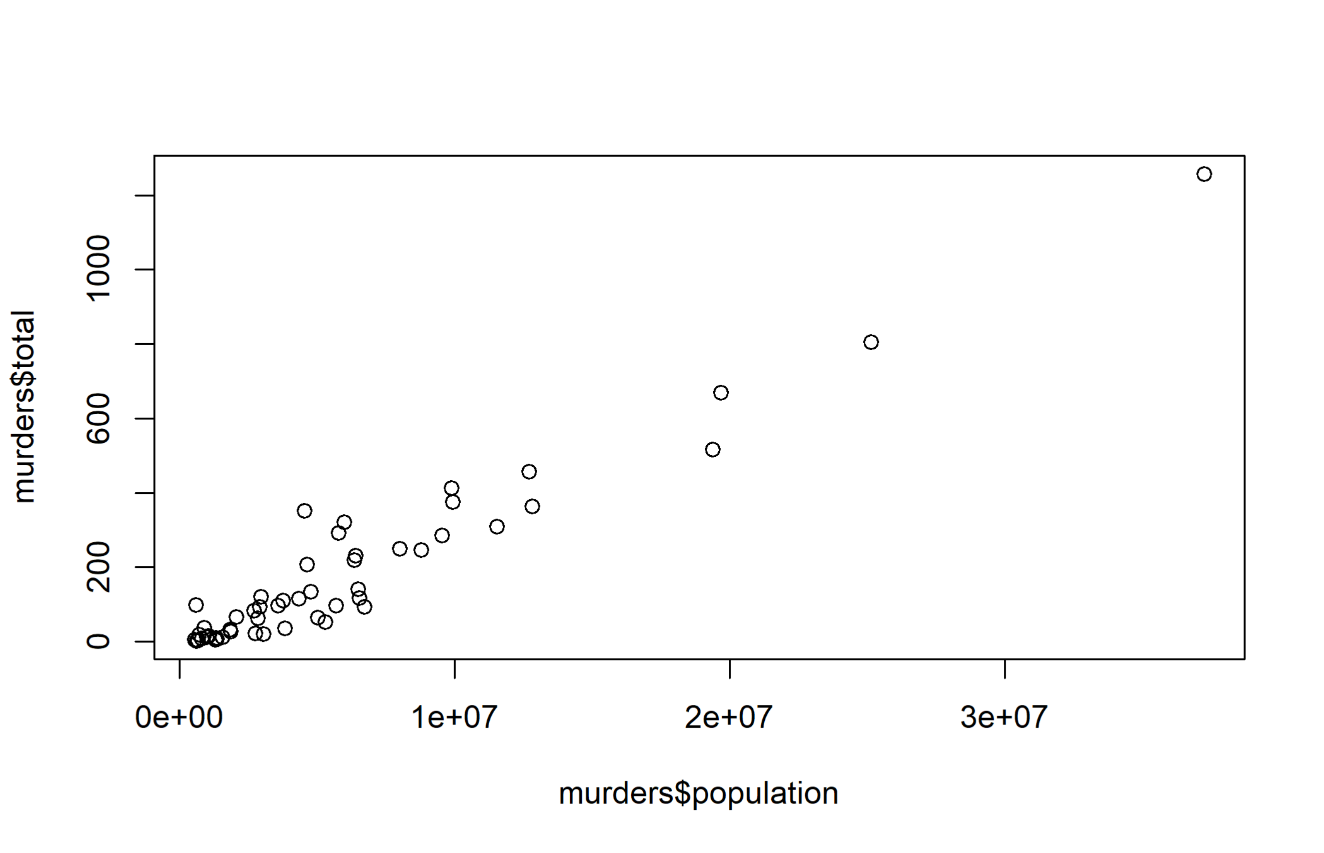
```
hist(murders$total)
```



Basic plots

Make a quick **scatterplot:**

```
plot(x = murders$population, y = murders$total)
with(murders, plot(x = population, y = total))  # These lines are equivalent
```



Interlude

Cleaning up

You *could* remove objects from your environment (R's memory) using `rm`:

```
a = "hi"
rm(a)
```

But generally it's better to just **start a new R session.** (Try this now.)

- Your environment is transient. Don't get attached to objects in it.
- Exit R when you're done working. Never save your environment.
- To re-create objects later, plan to re-run your script.
- When you need to keep something, save it to a file (we'll get to this soon).

Download these slides

Link: github.com/msu-econ-data-analytics/course-materials

Try to keep typing all the code yourself. **But also open these slides** in case you temporarily fall behind or want to go back to a previous slide yourself.

These slides are written in R Markdown (.Rmd file), which we'll cover in a couple weeks. You can look at either the .html (slides) or .Rmd (source) file.

- I like to create my own "reference script" where I collect all the new functions I'm learning and annotate/comment them as I go.

Vectors

Vectors

Vectors are the most basic objects in R. `a = 1` produces a vector of length 1.

To create longer vectors, use `c()`, for "concatenate":

```
codes = c(380, 124, 818)
countries = c("italy", "canada", "egypt")
class(codes)
```

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

```
class(countries)
```

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

In R, you can use either single or double quotes:

```
countries = c('italy', 'canada', 'egypt')
```

Why doesn't it work to type `countries = c(canada, spain, egypt)`?

Names

We can name the entries of a vector (with or without quotes):

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

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

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

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

Or by using the `names` function:

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

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

Sequences

Another useful way to create vectors is to generate sequences:

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

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

Shortcut for consecutive integers:

```
1:10
```

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

Counting by 5s:

```
seq(5, 50, 5)
```

```
##  [1]  5 10 15 20 25 30 35 40 45 50
```

Subsetting/Indexing

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
```

Sequences are useful if we want to access, say, the first two elements:

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

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

Subsetting/Indexing

You can also index using names, if they're defined:

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

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

And you can assign new values to indexed elements:

```
codes[2] = 125
codes
```

```
##  italy canada  egypt
##    380    125    818
```

Challenge

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

Change the name of the column "total" to be "murders", and then change it back to "total". (Hint: use `names()` and indexing.)

Converting (coercing) types

Turn numbers into characters, and back again:

```
x = 1:5
y = as.character(x)
y
```

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

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

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

Converting (coercing) types

A vector can't mix and match types, so R will just guess:

```
z = c(1, "canada", 3)
z
```

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

```
class(z)
```

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

If a conversion isn't obvious to R, you'll get an `NA` ("not available"):

```
as.numeric(z)
```

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

Special values

In R, `NA` contains no information.

```r
NA == NA
```

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

```r
NA + 0
```

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

```r
is.na(NA + 0)
```

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

`NA` values are very important in representing missing data.

Special values

Other special values in R:

```r
1/0
```

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

```r
-1/0
```

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

```r
0/0
```

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

Vector arithmetic

Arithmetic operators apply **element-wise.**

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

Multiply a vector by a scalar:

```
inches = 1:12
cm = inches * 2.54
cm
```

```
##  [1]  2.54  5.08  7.62 10.16 12.70 15.24 17.78 20.32 22.86 25.40 27.94 30.48
```

Divide (the elements of) one vector by (the elements of) another:

```
murder_rate = murders$total / murders$population * 1e5
mean(murder_rate)
```

```
## [1] 2.779125
```

An aside on data frames

We could add the murder rate to our data frame as a new variable (column):

```
murders$rate = murders$total / murders$population * 1e5
head(murders)
```

```
##        state abb region population total     rate
## 1    Alabama  AL  South    4779736   135 2.824424
## 2     Alaska  AK   West     710231    19 2.675186
## 3    Arizona  AZ   West    6392017   232 3.629527
## 4   Arkansas  AR  South    2915918    93 3.189390
## 5 California  CA   West   37253956  1257 3.374138
## 6   Colorado  CO   West    5029196    65 1.292453
```

But this isn't always the best approach to editing data frames. Why?

- The syntax is redundant and gets complicated quickly.
- It directly modifies your original data frame, rather than creating a new version.
- If there is already a column named `rate`, it gets overwritten.

An aside on data frames

One potentially better approach uses `cbind` to create a new object:

```
murders_with_rate = cbind(murders, murder_rate)
head(murders_with_rate)
```

```
##        state abb region population total murder_rate
## 1    Alabama  AL  South    4779736   135    2.824424
## 2     Alaska  AK   West     710231    19    2.675186
## 3    Arizona  AZ   West    6392017   232    3.629527
## 4   Arkansas  AR  South    2915918    93    3.189390
## 5 California  CA   West   37253956  1257    3.374138
## 6   Colorado  CO   West    5029196    65    1.292453
```

What should you make sure to watch out for when using `cbind`?

Subsetting with logicals

It's often useful to **subset** a vector based on the properties of another vector.

Generate a logical vector that says whether each element of a vector passes a test:

```
low = murder_rate <= 0.6  # this is "< =" without a space
low
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [49] FALSE FALSE FALSE
```

Subsetting with logicals

Now we can subset (index) states using this logical:

```
murders$state[low]
```

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

How many states meet this test? `sum` coerces logical to numeric, treating `TRUE` as 1 and `FALSE` as 0:

```
sum(low)
```

```
## [1] 4
```

Challenge

Try this on your own, and compare with a neighbor:

Which state has the most murders?

Hint: Use logical indexing and the `max` function.

Miscellaneous basics

A useful trick: %in%

Is Montana listed as a state in this dataset?

```r
"Montana" %in% murders$state
```

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

How about D.C. and Puerto Rico?

```r
c("District of Columbia", "Puerto Rico") %in% murders$state
```

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

Lists

Lists are objects that can store any combination of types.

```
record = list(
  name = "John Doe",
  id = 1234,
  grades = c(94, 88, 95)
  )
record
```

```
## $name
## [1] "John Doe"
##
## $id
## [1] 1234
##
## $grades
## [1] 94 88 95
```

FYI: A data frame is a list of vectors that follows certain rules.

Lists

Access the components with `$` as usual, or with double square brackets:

```
record$id
```

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

```
record[["id"]]
```

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

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

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

```
record$grades[3]
```

```
## [1] 95
```

Table of contents

1. About R
2. Operators
3. Objects and functions
4. Data frames
5. Vectors
6. Miscellaneous basics